LIFTING THE ABSTRACTION LEVEL OF COMPILER TRANSFORMATIONS

A Dissertation

by

XIAOLONG TANG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jaakko Järvi
Committee Members,	Bjarne Stroustrup
	Gabriel Dos Reis
	Frank Sottile
Head of Department,	Duncan M. (Hank) Walker

August 2013

Major Subject: Computer Science and Engineering

ABSTRACT

Production compilers implement optimizing transformation rules for built-in types. What justifies applying these optimizing rules is the axioms that hold for built-in types and the built-in operations supported by these types. Similar axioms also hold for user-defined types and the operations defined on them, and therefore justify a set of optimization rules that may apply to user-defined types. Production compilers, however, do not attempt to construct and apply these optimization rules to user-defined types.

Built-in types together the axioms that apply to them are instances of more general algebraic structures. So are user-defined types and their associated axioms. We use the technique of generic programming, a programming paradigm to design efficient, reusable software libraries, to identify the commonality of classes of types, whether built-in or user-defined, convey the semantics of the classes of types to compilers, design scalable and effective program analysis for them, and eventually apply optimizing rules to the operations on them.

In generic programming, algorithms and data structures are defined in terms of such algebraic structures. The same definitions are reused for many types, both built-in and user-defined. This dissertation applies generic programming to compiler analyses and transformations. Analyses and transformations are specified for general algebraic structures, and they apply to all types, both built-in and primitive types.

DEDICATION

To my daughter, Olivia Rui Tang.

ACKNOWLEDGMENTS

First, I want to thank Dr. Järvi, my adviser. Without him, I would not have become what I am today. Throughout my entire PHD study, Dr. Järvi has been ensuring me a worry-free study and research environment; in particular, he has been giving me enough freedom to pursue my interests in programming languages and language processing. As my academic adviser, he has been putting a great deal of efforts for having me to be a professional writer and researcher. To ensure that my research is on the right track, he has been providing timely advices and guidelines to me. Besides, he has been encouraging me to reach out to communicate with other researchers.

Second, I want to thank Dr. Dos Reis. His classes further exposed me to various topics on programming languages and language processing. And he has been able to provide clear answers to my questions. As a matter of fact, Dr. Dos Reis has been making me to be more and more interested in programming languages and language processing.

Next, I want to thank Dr. Stroustrup and Dr. Sottile. As a figure in the field of programming languages, Dr. Stroustrup is also my icon. He is a real example of how working on programming languages could have a large positive effect on our life. Talking with Dr. Stroustrup is inspiring and full of fun. Dr. Sottile is my fourth committee member. His questions have been piquing; these questions shall be answered before my research can be put into practice in a large scale.

I am also grateful to the administration staff in my department, including but not limited to Kay Jones and Tina Broughton. They have been always ready for providing help with some administration issues for me.

Finally, I want to thank my family; I owe a lot to them. The love from my mother and father has been accompanying me in my entire life. They have been ready for providing any support for my success. My wife has been experiencing my

happiness and sadness along my way. My little daughter is the source of my courage and confidence.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

FIGURE Page

1. INTRODUCTION

Optimizing is a process of semantics-preserving transformations. The behavior of an optimizer is defined by the set of analyses and transformations this optimizer supports. How broadly the transformations apply has a significant impact on the effectiveness of an optimizer. This dissertation focuses on broadening the applicability of common optimizations to user-defined types and operations and thus on improving the effectiveness of compiler transformations for imperative programming languages.

1.1 Abstraction Penalties

Implementing a practically useful compiler for a general purpose programming language is expensive. To manage the complexity of such a task, a compiler is usually divided into three modules—the front-end, the middle-end, and the back-end. The front-end of a compiler is designed to be language specific; it translates a program into its AST (abstract syntax tree). The back-end is target specific; it emits code (instructions in binary form) for specific targets. The middle-end is independent of languages and targets; it usually employs a language- and target-neutral IR (intermediate representation) and performs analyses and optimizations upon it.

That the middle-end is language- and target-neutral is important for reuse and modularity. At the same time, it is, however, a challenge for the optimizer. The IR of the middle-end of a compiler generally does not maintain information about user-defined types; user-defined types are mainly used for type checking source code in the front-end and discarded afterwards. As a result, the middle-end does not directly support optimizations for user-defined types. Instead, transformations in the middle-end are defined only for built-in types and the set of built-in operations for them. User-defined abstractions are thus obstacles to compiler optimizers; compilers merely

translate them into lower-level representations according to their definitions before attempting to perform optimizations.

The difficulty of optimizing user-defined abstractions raises a performance concern with the use of abstractions. On the one hand, abstraction helps solve problems by reducing the complexity of problems, and indeed high-level programming languages provide rich facilities for defining abstractions. For example, C++ allows extending the syntax of built-in types to user-defined types, by operator overloading. As a result, a programmer may use concise notations for expressing computations involving user-defined types. On the other hand, uses of abstractions come with costs, such as function call overhead and increased number of temporaries.

To quantify the impact of abstractions to performance, Stepanov defines *abstraction penalty* as the ratio of the execution time of a high-level, abstract implementation over that of the corresponding low-level, direct implementation [1, §D.3]. A concrete example of abstraction penalties follows.

Let `x`,`y`, and `z` be of some user-defined vector type that supports vector arithmetic. Consider the expression `z = x + y`. A compiler likely translates this expression into two successive simpler expressions, `t = x + y; z = t`, where `t` is a temporary vector variable, introduced by the compiler. If the above expressions would be for integers, or for another built-in numeric type, a compiler would be able to eliminate the effect of the additional temporary; it would apply *copy propagation* to replace the use of `z` with the use of `t`, and it would rely on *unreachable code analysis* to remove `z = t`. Similar optimizations for a user-defined vector type, however, are beyond the capabilities of production compilers. Optimizing away `z = t` for vectors would require a compiler to have semantic knowledge about the operations on vectors, including that the assignment operator creates two objects that compare equal. A compiler typically neither maintains nor takes advantage of such knowledge. As a result, in the case of vectors, `z = t` cannot be directly optimized away by a compiler. At most a compiler translates the expression into its low level representation according to

the definition of vector assignment operator, which typically involves a loop, and then attempts to perform optimizations. Such a way of optimizing user-defined abstractions is opportunistic—the effectiveness of optimizations depends on many arbitrary factors.

To see why a compiler has difficulty in optimizing operations involving user-defined types, we analyze another simple example. Its simplicity is a bit deceiving, as the interplay of many analyses and optimizations are needed for a not particularly impressive transformation. Consider the code in Figure 1.1. Assuming a few

```
T x, y, z, w, r, s;
... // initialization
r = x + y + z;
... // code that does not change x and y
s = x + y + w;
```

Figure 1.1. A code sample containing common subexpressions.

conditions on the type T, that copying produces two distinct objects that compare equal and that operator+ has no side-effects, this code contains several optimization opportunities. These opportunities become more evident when the compiler breaks complex expressions into simpler ones as illustrated in Figure 1.2. If *common*

```
T x, y, z, w, r, s, t1, t2, t3, t4;
... // initialization
t1 = x + y;
t2 = t1 + z;
r = t2;
... // code that does not change x and y
t3 = x + y;
t4 = t3 + w;
s = t4;
```

Figure 1.2. Code resulting from simplifying the code in Figure 1.1.

subexpression elimination is applied, the expression `t3 = x + y` becomes `t3 = t1`. This rewrite reveals another optimization opportunity, namely that of *copy propagation*: the use of `t3` in `t4 = t3 + w` can be replaced with `t3`'s definition `t1`. Eventually `t3` becomes unused, and the compiler can elide its definition altogether as dead code.

The above reasoning can only be expected of a typical modern compiler if `T` is some built-in type, such as `int`. If `T` is a user-defined type it is unlikely that the optimizations are performed, even if `T` is a simple "value type," such as one representing a coordinate pair or a complex number. If a compiler does not know the semantics of a user-defined type's operations, it is forced to make conservative assumptions. E.g., to show that `x` and `y` cannot change between the two calls `x + y` is beyond most compilers' abilities.

In general, even though the operations and properties of a user-defined type were essentially the same as those of some built-in type, a compiler is not capable of the same class of analyses and optimizations for the user-defined type as it is for built-in types.

1.2 Generic Transformations

Extending a compiler to apply its built-in optimizations to some particular user-defined type would be useful for that particular type but applying this approach on a type-by-type basis quickly becomes unmanageable, leading to a proliferation of compiler versions; a large monolithic compiler that would perform optimizations for all possible types is not realistic. The cost of maintaining multiple versions of compilers is high and different compiler versions are not composable. For example, given one compiler that supports optimizations for the `std::vector` class and one for the `std::list` class, it is not straightforward to build a compiler that supports optimizations for both of the two classes.

This dissertation develops more economical means to enable traditional optimizations for user-defined types. In this thesis, we rely on *generic programming*. This

programming paradigm, advocated by Stepanov and others, achieves significant code reuse through systematic categorization of abstractions. Analogously, such a systematic categorization can be the basis of generic and reusable compiler optimizations. This is natural as programmers of generic code routinely rely on the algebraic properties of types to transform their code (manually). In this respect, this dissertation is a work towards fulfilling the goal described by Dehnert and Stepanov [2]:

> Ultimately, we would like compilers to be able to perform such optimizations [common subexpression elimination, const and copy propagation, and loop-invariant code hoisting and sinking] at a high semantic level as well as they do at the built-in type level.

1.3 Domain-Specific Optimizations

Even if we succeed in constructing a compiler that is capable of traditional optimizations for user-defined types, we should not stop there. Modern software construction uses off-the-shelf libraries heavily; an application is usually built upon a set of reliable libraries. Such library-centric software construction may not arrive at efficient code, however. Two reasons are identified for this.

First, a library usually encapsulates domain knowledge, e.g., algebraic properties of types. Such knowledge is an opportunity for further optimizations. It is often the case that two operations may be semantically equivalent but one is more readable and the other is more efficient. To transform the former to the latter requires domain knowledge, but such domain knowledge is rarely exploited in a production compiler. Languages do not provide mechanisms to convey such knowledge to the compiler.

Further, the performance of a library may depend on the context where this library is used, including the run-time system, the operating system, and the architecture; taking into account the context information may allow a compiler to generate specialized, highly efficient code for the library in a particular context. As there

are typically no mechanism to communicate such context knowledge to a compiler, obtaining the most performance when composing libraries may be challenging.

To address performance concerns with the use of libraries, this dissertation studies how to support domain-specific optimizations in compilers.

1.4 Infrastructure for Generic Optimizations

The goal of this dissertation is to enable the implementation of reusable and extensible transformations in a compiler. Reusable in the sense that transformations apply to user-defined types and built-in types equally and that a single implementation of an optimization applies to more than one type. Extensible in the sense that a compiler shall allow a programmer to add custom transformations. To achieve its goals of reusability and extensibility, this dissertation studies the compiler infrastructure for supporting reusable transformations, a variety of reusable transformations, language support for user-defined transformations, and the necessary program analysis support.

Specifically, this dissertation describes an infrastructure for implementing *generic* compiler optimizations that apply to open-ended classes of user-defined (or built-in) types. Transformations on operations of particular types are instances of these generic optimizations. This infrastructure follows the principles of generic programming, a paradigm for designing and implementing reusable software libraries. It was popularized to the mainstream by the Standard Template Library (STL) [3], now part of C++ standard library [4]. Following the example of the STL, many successful generic libraries have been developed for a variety of domains [5–11]. Characteristic to these libraries are rigorously specified interfaces, using "concepts." Concepts are essentially algebraic descriptions of requirements on types—both syntactic and semantic. In generic libraries, generic algorithms are expressed in terms of abstract properties of types using concepts, rather than concrete types, and data types are

categorized according to which abstract properties they satisfy, i.e., which concepts they *model*.

For reusable, extensible optimizations, this infrastructure enables transformations to be defined in terms of concepts. This capability is obtained by supplying a traditional compiler with additional rich type information, including types of variables and functions, concepts, and the modeling relations from types to concepts. Such rich type information is accessible throughout the compiler pipeline in the infrastructure.

Type systems of mainstream programming languages do not typically allow the expression of semantic properties. Particularly, in C++, the language this thesis uses as a study subject, concepts do not have an explicit representation in the type system; they are a documentation convention. To improve C++'s support of generic programming, recent extensions to C++ lift concepts from documentation to language features [12]. This extended language provides an experimental platform for the work in this dissertation, and this dissertation refers to it as ConceptC++.

Note that the concepts proposal was decoupled from the C++11 standard. There were a variety of uncleared worries and concerns about using the proposed concepts facilities at the time of deciding on the features of the C++11's standard. However, previous work on concepts has showed the technical strength of using concepts in generic programming. As Stroustrup pointed out [13],

> "Concepts," as developed over the last many years and accepted into the C++0x working paper in 2008, involved some technical compromises (which is natural and necessary). The experimental implementation was sufficient to test the "conceptualized" standard library, but was not production quality. The latter worried some people, but I personally considered it sufficient as a proof of concept.

There is an ongoing effort [13] to introduce concepts into a future revision of C++, led by Stroustrup.

This dissertation views the concepts feature as a non-intrusive means for communicating knowledge of abstractions (including transformation rules) to compilers. Without the concepts feature in C++, other kinds of annotations could be used. This dissertation uses the concepts features, but the exact form of how concepts are expressed is not essential. What is essential is a means to categorize types and attach algebraic rules to these categories.

1.5 Contributions

This dissertation studies building reusable, extensible transformations in modern compilers for mainstream imperative programming languages, C++ in particular. As its contributions, this work

- Proposes "concept-based optimization" to support extensible optimizations and develops a framework for supporting user-defined domain-specific transformations. The framework is based on term rewriting. In particular, this dissertation identifies a set of algebraic structures and their algebraic properties relevant for typical compiler transformations. The framework allows to recognize models of these structures and apply transformation rules derived from these structures to an open-ended set of user-defined types.

- Devises an approach that extends generic term rewriting beyond the front-end of a compiler into the middle-end. This approach utilizes the compiler's existing functionalities to translate rewrite rules and subject code side by side into their respective intermediate representations. For effective rewriting, this approach assigns an *abstraction index* to a function or a rewrite rule for indicating the relevant abstraction level where a function or a rewrite rule stays among all functions and rewrite rules in a program, and uses the abstraction indices to address the phase ordering problem arising from combining term rewriting and function inlining.

- Implements a concept-based optimizer prototype, as an extension to the ConceptGCC compiler [12]. It exploits the "concepts" language feature [14] of the C++ concepts proposal [15] for specifying generic rewrite rules and justifying their validity for desired user-defined types.

- Evaluates concept-based optimizations with some customized code and a micro-benchmark from Adobe [16]. The evaluation results indicate that the concept-based approach can effectively eliminate abstraction penalties in C++ programs.

- Explores the structure of composite objects, one important class of types in generic programming, and characterizes the aliasing invariant of composite objects. It then presents a summary-based program analysis that uses the properties of composite objects to reduce analysis efforts and improve analysis precision.

- Evaluates the program analyzer by analyzing uses of the STL containers and three real-world applications. This evaluation shows that the analyzer produces more precise and concise procedure summaries. Compared to traditional analysis, we measure significantly smaller points-to relations and procedure summaries in analyzing real-world applications.

- Lifts a few classes of optimizing transformations into generic transformations in the concept-based framework. These generic transformations apply equally to user-defined classes and built-in types.

- Prototypes generic transformations in the LLVM infrastructure [17] and the Clang compiler [18]. These generic transformations are integrated with the above program analyzer.

- Evaluates the transformation prototype with SPEC2000 [19]. The capability of optimizing user-defined types gains modest performance speedup for SPEC2000.

1.5.1 Optimization Cases

To give a concrete example of the optimization capability attained in this work, consider the code below.

```
int accumulate(const std::vector<int> &grades) {
  int sum = 0;
  for (std::vector<int>::iterator iter = grades.begin(); iter != grades.end(); iter++) {
    sum += *iter;
  }
  return sum;
}
```

The above code simply iterates over all elements of a vector, the input of the accumulate function, and accumulates the values of these elements. It is obvious to a programmer that the method call grades.end() is loop invariant— this call on each iteration produces the same side effect, e.g., accessing the same memory with the same value each iteration and/or writing to the same memory the same value each iteration. This optimization opportunity, however, fails to be revealed by a traditional optimizing compiler.

The foremost challenge for optimizing the above code is efficiently and effectively computing the aliases arising from the use of pointers (and references). Such pointer analysis is usually prohibitive to a traditional compiler, requiring being flow-sensitive, context-sensitive, and inter-procedural. This thesis attacks the challenge by utilizing the knowledge of user-defined types, e.g., the aliasing invariant with the vector<int> class, to make pointer analysis affordable. Our approach manages to reveal the loop-invariant method call in the above code.

In addition to hoisting grades() out of the loop, there are other opportunities for high-level optimizations. For example, in our approach we can augment the compiler with the knowledge of a single generic rule such as iter++ $\rightarrow$ ++iter which can allow

replacing the use of iter++ with ++iter (++iter is often more efficient than iter++ in C++).

1.6 Thesis Organization

The structure of the thesis is as follows. After introduction, Chapter 2 reviews related work. Chapter 3 overviews the approach of exploring concepts for optimizations and describes the background on the approach. Chapter 4 presents an approach that exploits concepts and axioms for enabling generic, user-defined transformations in the compiler. Transformations described in Chapter 4 are limited to the front-end of the compiler; Chapter 5 extends these transformations into the middle-end of the compiler. Chapter 6 describes how to exploit the compositeness property of user-defined types to improve the precision of program analyses. Relying on the analysis in Chapter 6, Chapter 7 studies lifting some common optimizations into generic ones such that they apply to user-defined types as well. Lastly, Chapter 8 concludes this thesis and discusses future work.

2. RELATED WORK

This dissertation pushes a traditional optimizing compiler to take advantage of the semantic and domain knowledge of user-defined abstractions for optimizations. In a broad sense, this work is related with using abstractions in software construction, and, specifically, with the areas of program analyses and program transformations.

2.1 Library-Centric Software Construction

A programming language is designed for facilitating solving problems. To solve a given problem, it is appealing to use a special-purpose language which provides primitive operations for performing computations in the domain of this problem. A special-purpose language can allow for a concise and intuitive implementation, for which a compiler of the special-purpose language may ensure high performance.

Developing and maintaining the toolset (the compiler, debugger, linker, and run-time support) for a special-purpose language is, however, expensive, considering a limited number of users. In contrast, the analogous cost for a general-purpose programming language, like C, C++, or Java, is amortized over a larger base of users.

To achieve the effect of a special-purpose language, a general-purpose language may be extended with domain-specific features. For example, Rex Jaeschke formed the numerical C extension group for extending C with linguistic features for scientific computing [20]. Supporting domain-specific features with language extensions is, however, controversial. Language extensions require compiler support, which compiler vendors tend to be reluctant to provide until the extensions become part of language standards. At the same time, including a language extension into a language standard receives cautious considerations. To be included, a language extension should be of interest to a large number of users and not increase the language's complexity disproportionately. Consequently, language extensions are a limited success in providing domain-specific features to general-purpose programming languages.

Libraries are alternatives to language extensions for adding domain-specific support for general-purpose languages. A well-designed library can provide a coherent and complete set of data types and operations for computations within a particular domain. As an example, consider the domain of linear algebra. Early in 1973, Hanson, Krogh and Lawson argued about the benefits for using a set of basic routines for solving problems in linear algebra [21]. Since then, the basic linear algebra subprograms(BLAS) [22–26] have been successful libraries in FORTRAN for building high-quality linearly algebra software, e.g., LINPACK [27]. The success of BLAS in FORTRAN has been replicated in other languages. CBLAS, a C library, provides a set of interfaces which are equivalent to the interfaces BLAS defines [28]; uBLAS makes BLAS functionality available to C++ users [29].

Today libraries are central in building applications; applications are built on reliable libraries. Often there are conflicting concerns for library design. Providing concise, convenient notations or safety guarantees may compromise performance, for example. Thus, some libraries provide multiple semantically equivalent sets of interfaces for different goals. For instance, a programmer may use the overloaded subscript operator to access an element of one instance of the vector template in the standard template library (STL) [3]; this operation is not safe in that the subscript operator performs no checks. In case that safety is valued above performance, a programmer may choose to use the `at` method of the STL vector to access elements; the `at` method performs run-time checks. Another example is LiDIA, a C++ library for computational number theory [30]. For readability and maintainability, this library provides a set of interfaces following convenient and familiar mathematical notations. At the same time, it provides an alternative set of interfaces which are less convenient but promise better performance. With such dual interfaces, libraries can satisfy different needs of users, but add complexity to their use, bothering users with questions about interface choices.

A programming paradigm of particular interest for library design is generic programming. Generic programming promises reusable and efficient libraries. Reusable algorithms and components are obtained through parametric polymorphism and efficiency is achieved by template specialization and compile-time overload resolution. Examples of generic libraries include BGL [5, 11], MTL [6], and STAPL [9].

Though C++'s way of implementing parametric polymorphism using templates is efficient, use of well-designed generic libraries does not necessarily result in efficient applications; composing libraries is challenging, as described in Section 1.3. Several approaches have been proposed for addressing this issue. One idea is generative programming and active libraries [31]. Generative programming resembles generic programming in that they both aim for reusable and efficient software products. Their strategies for enhancing software construction, however, are different. Generative programming is a software engineering paradigm; it is concerned with all phases of software development; it models families of abstractions such that the differences between them are encapsulated in customizable features, and a user is allowed to choose the right set of features for an efficient, specialized component in a particular context. In contrast, generic programming is a paradigm for designing libraries; it seeks the commonality of similar implementations of a single algorithm, and extracts the commonality into abstractions (concepts) such that a single, generic implementation of this algorithm can cover many concrete implementations.

Libraries designed following generative programming are often called *active libraries*. An active library provides not only a set of abstractions, but a set of optimizations for the abstractions as well. Examples of active libraries include MTL [6], POOMA [32], and Blitz++ [33]. At compile time, an active library plays an active role in generating efficient code by benefiting from the set of optimizations defined in this library. For example, to make full use of the cache line of an architecture, a loop in the body of the `transpose` method in MTL is automatically unrolled. Though active libraries are often easy to adopt in that they require no modification to exist-

ing compilers, the techniques that are used for implementing active libraries require quite some expertise. Further, the transformations that active libraries support are limited to the front end of the compiler and sophisticated analyses are not used.

2.2 Program Transformations

Besides active libraries, several approaches have been designed for addressing the performance issues with using libraries. They differ in implementation strategies, usability, effectiveness, scalability, and economics. This section reviews them, categorized according to their implementation strategies.

Reflection and metaprogramming Reflection and metaprogramming allow a programmer to manipulate the behavior of a program. With Java or C# reflection, a programmer may examine or modify the run-time state of an application, e.g., by enumerating the methods which a class defines and invoking these methods with an object of this class. As opposed to reflection, metaprogramming is for manipulating programs at compile time rather than at run time.

Several works explore reflection and metaprogramming for domain-, application-, or even class-specific transformations. The aforementioned active libraries are implemented by metaprogramming. In particular, Todd Veldhuizen and David Vandevoorde reveal that C++ expression templates are an effective means to eliminate temporary variables for expressions operating on user-defined types [34–36]. Another work is OpenC++ [37]. OpenC++ is an extended version of C++. In OpenC++, an object may be associated with a metaobject; such an object is called as a reflective object to distinguish itself from a normal object. The behavior of a reflective object may be altered by the metaobject associated with it. For example, invoking a method of a reflective object may be intercepted, such that additional operations may be performed before entering into and/or exiting from this method.

Reflection and metaprogramming are advanced features for program manipulation. They do not come without cost. Metaprogramming demands expert skills, while reflection requires run-time support and can have a notable performance overhead.

Term rewrite system Rewrite rules are effective for communicating user-defined transformations to compilers. User-specified rewrite rules for optimizations are supported even in some familiar popular programming languages such as in Haskell [38]. Next we describe a few works that exploit term rewriting for domain-specific transformations, and are relevant for our work. We start from the most closely related works.

Sibylle Schupp et al. designed Simplicissimus for exploiting domain-specific knowledge for optimizing transformations in C++. Simplissimus is a source-to-source translator [39]; it applies user-defined transformation rules during the course of translation. What most characterizes Simplissimus is the means of defining rewrite rules and the preconditions on applying them. In Simplissimus, a rewrite rule and the constraints on applying this rewrite rule are grouped into a class template. Such a rewrite rule in a class template applies to a type if this type is specified to satisfy the constraints this class template defines.

Simplicissimus follows the principles of generic programming. In Simplicissimus, rewrite rules in a class template are generic; they apply to an open-ended set of types. This dissertation shares with Simplicissimus the same inspiration that generic programming can support extensible transformations. What differentiates the two works is as follows. First, Simplicissimus uses C++ template metaprogramming for defining and effecting code transformations while this dissertation leverages the linguistic support for concepts in ConceptC++. Moreover, this dissertation extends the applicability of generic transformations from the exact syntactic pattern matching that Simplicissimus requires.

Bagge and et al. also exploit term rewriting for domain-specific optimizations. Their work, called CodeBoost [40], is built upon a general term rewrite language, called *Stratego* [41, 42]. Stratego provides abstractions for users to define strategies for applying rewrite rules. A *strategy* may be as simple as an identity transformation, or it may be a complex composition of different strategies. In CodeBoost, a rewrite rule is prefixed with a strategy that controls the rule's application. Besides leveraging Stratego, CodeBoost enhances the expressiveness of rewrite rules in several aspects. First, rewrite rules are defined in C++ syntax; variables involved in rewrite rules may be declared with types and such type information is taken into account for distinguishing overloaded operators. Second, rewrite rules may be optionally associated with conditions; these conditions act as guards for applying the rewrite rules. Third, rewrite rules may be defined independently of types and function signatures, thus being generic and applicable in many scenarios.

Following the line of work of building domain-specific transformation in a general term rewrite language, Visser explores dynamic rewrite rules for overcoming the limitation that term rewriting is context free [43]. Dynamic rewrite rules are generated on the fly at run time to propagate data-flow facts. Visser describes that combining scope information with dynamic rewrite rules, i.e., "scoped dynamic rewrite rules", is expressive enough for defining transformations such as function inlining. Olmos et al. further explore dynamic rewrite rules for expressing more general transformations, such as cost propagation, copy propagation, and common subexpression elimination [44]. The key to their work is introducing "dependent dynamic rewrite rules" to model the fact that dynamic rewrite rules depend on data-flow information. Dependent dynamic rewrite rules eases maintaining the mapping from dependencies to the rewrite rules they affect, allowing certain transformations to be specified concisely.

Rewrite rules have of course been exploited for transformations from early on. For example, TAMPR, initiated in 1970, is a rewrite-rule based transformation system

[45] that has been successfully used to define high-level transformation rules for calls to the routines of the LINPACK library.

As in Schupp and et al.'s work [39], user-defined rewrite rules, in this thesis, are in C++ syntax and they are generic. Also as in Bagge and et al.'s work [40], rewriting, in this thesis, is flow-sensitive—conditions are associated with rewrite rules to express transformations that are dependent on data-flow facts. What distinguishes this thesis from them is that we aim for building a general framework that allows to exploit common knowledge of data types for transformations throughout the whole compilation pipeline. In this framework, we use ConceptC++ to concisely express type-safe generic rewrite rules; we exploit the common properties of data types for an affordable, scalable, and precise pointer analysis, to enable a broad range of transformations for user-defined types, including conducting flow-sensitive rewriting.

Annotation languages and sophisticated analysis frameworks Employing sophisticated analysis, e.g., for computing data-flow facts or reasoning about the semantics of user-defined functions, may reveal more transformation opportunities than relying on traditional analyses in compilers. Such sophisticated analysis is challenging, however. Precise data-flow analysis is impossible or expensive for practical programming languages. Therefore, some works utilize annotation languages for communicating the semantics of user-defined abstractions to compilers hoping to enable sophisticated analyses and transformations.

Guyer et al. provide a compiler, called Broadway, for exploiting domain-specific transformations. Broadway applies abstract interpretation for uncovering transformation opportunities [46]. Abstract interpretation maps concrete objects to abstract objects and simulates the execution of a program in the domain of abstract objects [47]. The goal (of abstract interpretation) is to gain some information about the actual running of a program. The necessary knowledge for enabling abstract interpretation is the abstract domains and the abstract semantics of functions oper-

ating in the abstract domains. Broadway designs an annotation language for users to provide such knowledge to the abstract interpretation engine.

Kennedy et al. exploit axiomatic systems for reasoning about programs and discovering transformation opportunities [48–50]. Their work, called *Telescoping languages*, also relies on annotating functions to describe their semantics. So, like Broadway, the telescoping languages approach puts the annotation burden upon users. Compared with Broadway and Telescoping languages, this thesis reduces the annotation efforts through generic transformations and performs sophisticated data-flow analysis for automatically reasoning about data-flow facts of programs.

Declarative specification languages Extensible, or *open*, compilers are one possible approach for supporting domain-specific transformations. An open compiler allows users to inject custom transformations to the compilation process. This is usually achieved with declarative specification languages.

One example of an open compiler is the Cobalt project by Lerner et al. [51]. This project develops a language for specifying transformations and the analyses which these transformations depend on. A noteworthy feature of Cobalt is that it produces automatic soundness proofs of transformations defined in it. Recently, Willcock proposed a regular expression language called *Pavilion* for an extensible compiler [52]. Like this thesis, Pavilion also applies the idea of generic programming for its extensible transformations.

2.3 Program Analyses

This thesis draws from many works in the area of program analysis. Program analyses reveal information about possible run-time states of a program. Such information is usually required for justifying program transformations. Typically, a compiler performs data-flow analysis for reasoning about the use and definition of variables. This data-flow information is the input to some other analyses and some

transformations, e.g., reaching definition analysis, liveness analysis, available expression analysis, and constant propagation [53]. Precise data-flow analysis is challenging, however. In particular, the aliasing issues arising from the use of pointers complicate data-flow analysis. On the other hand, given the points-to relations at every program point of a procedure, it is quite straightforward to compute the side effects that the procedure may produce. Assuming the availability of pointer analysis, this thesis performs side effect analysis according to the strategy described by Landi et al. [54]. This section next reviews works on aliasing.

Aliasing occurs when a storage is accessible in more than one way, usually as the result of the use of pointers. Alias analysis is to disambiguate aliasing by computing what pointers may refer to what objects. Landi proves that the aliasing problem is difficult; it is impossible to statically compute precise aliases for a programming language which supports if-statements, loops, dynamical storage, and recursive data structures [55]. Ramalingam provides different proofs for the same results as Landi's work [56]. Consequently, many works have focused on computing safe approximations to the aliasing problem.

A solution to the aliasing problem is *safe* if each possible run-time points-to relation at each program point is in the set of the points-to relations that this solution computes at the same program point. There exists a variety of safe aliasing analyses. These analyses differ in their precision and efficiency. Andersen describes a fast flow-insensitive analysis [57]. Flow-insensitive analysis does not take into account the order of program statements and thus may generate results that are too conservative. Andersen's approach derives subset constraints from the statements of a program; solving a constraint system composed of such subset constraints amounts to computing a safe approximation to the run-time points-to relations of the program. Steensgaard proposes a fast constraint-based pointer analysis [58], which is also flow-insensitive. Compared to Anderson's, his is faster but less precise.

Flow sensitivity is crucial for the precision of pointer analysis. Choi et al. present a flow-sensitive pointer analysis [59]. They formulate points-to analysis as a monotone data-flow framework [60], where an equation is established at each statement to describe this statement's effect on aliasing. Computing the fixed point solution for a set of such equations eventually produces a set of points-to relations or a set of aliases at each program point. Besides being flow-sensitive, Choi's approch is also interprocedural, context-sensitive, and field-sensitive. Interprocedural analysis takes into account the aliasing effects that function calls may produce; context sensitivity differentiates different calls to a single function; and field sensitivity distinguishes different member fields of objects. All of these techniques contribute to the precision of pointer analysis. This thesis applies these techniques for a precise points-to analysis. In addition, this thesis explores three further means to improve the precision and efficiency of pointer analysis: (1) type-based alias analysis, (2) summary-based analysis, and (3) aliasing control in object-oriented programming.

Type-based alias analysis Types are known to be helpful for disambiguating aliases. Diwan et al. discuss [61] three kinds of type-based alias analyses for detecting whether two objects may overlap. Type information alone may produce quite conservative results on aliasing. E.g., type-based analyses cannot determine that two objects of certain types are disjoint. This thesis observes that certain class invariant, that guarantee disjointedness is common and thus present in many classes. For example, two distinct objects of a STL string are disjoint. This thesis studies such classes and their class invariants, and explores the invariants in designing an efficient and precise points-to analysis.

Summary-base analysis Summary-based analysis can help scalability [62–64]. A procedure summary conservatively approximates a procedure, describing information such as the procedure's side effects and its impact on aliasing. Procedure summaries act as transfer functions at call sites. An analysis approximates the effect of a

function call by binding a calling context to the function's parameters in the callee procedure's summary; the body of the callee is not analyzed again.

Generating a precise procedure summary is usually not possible; the knowledge about a procedure's *invisible variables*, i.e., objects accessible via the procedure's parameters and via global variables used in a procedure [65, 66], is often incomplete. Wilson et al. argue that computing a procedure's summary based on all possible aliases of its invisible variables is prohibitively expensive [67]. Chatterjee et al. describe a summary based analysis for a simplified variant of C++ [62]. They use, in the context of object-oriented programming, the types of invisible variables to reduce the number of spurious aliases among the invisible variables: only if the type of one invisible variable is in a subtyping relation with the type of another invisible variable, the two variables may alias. This thesis exploits types and their properties to further reduce spurious aliases among the invisible variables. As a result, procedure summaries remain very small; they contain a small number of points-to relations and side effects.

Aliasing control Several works propose sophisticated type systems for enforcing aliasing control on objects, so that aliases may be better statically understood and verified. John Hogg introduces the notion of *islands* for ensuring that objects are not aliased [68]. An island encapsulates a set of objects and guarantees that accessing this set of objects must go through a unique interface. Thus, an island has no aliases. Almeida's balloon types are similar to island types, but the underlying implementation mechanism is different [69].

Island types and balloon types impose a full alias protection for an object in its entirety. In contrast, Noble et al. describe a more flexible partial alias protection strategy [70]. They use two parameters, *arg* and *rep*, in an object's type definition to specify which parts of an object are allowed to be referenced from outside and which are not. Clarke et al. subsequently design ownership types [71]. Ownership types

formalize the core of flexible alias protection. In a basic ownership type system, each object has a unique owner, which is another object or a predefined entity. Accessing an object must first access the owner of this object. In a followup work, Clark et al. explore ownership types for statically disambiguating aliases between objects [72].

Ownership type systems are quite relevant for the work presented in this thesis. This thesis studies how to exploit the knowledge for classes of data types for analyses and optimizations. One particular class of data types, regular composite types, studied in this thesis resemble ownership types. What distinguishes the thesis from ownership types is the way of exploiting type information for addressing the aliasing problem; this thesis performs program analysis to understand aliases while Clark's work uses type systems for imposing alias restrictions.

2.4 Summary

This section diagrammatically summarizes how this thesis is related with previous work. The two diagrams, Figure 2.1 and Figure 2.2, illustrate the related optimization techniques that previous works and this thesis propose for the front-end and middle-end of the compiler. In the two figures, the solid arrows denote normal processing flow in the traditional compiler and the dashed ones denote additional optimization flows explored by previous work and this thesis.

In Figure 2.1, normally source code and libraries are fed into the parser and type checker, which output the AST. The AST is then transformed into the intermediate representation (IR), e.g., via the lowering processing. In the normal processing flow, additional optimizations may be achieved by leveraging the capability of metaprogramming. Such examples include active libraries [31] and Simplicissimus [39].

To support domain-specific optimizations, third-party optimizers, such as Broadway [46], the Telescoping languages project [48], and CodeBoost [40], often work as source-to-source translators. For obtaining domain-specific knowledge, third-party

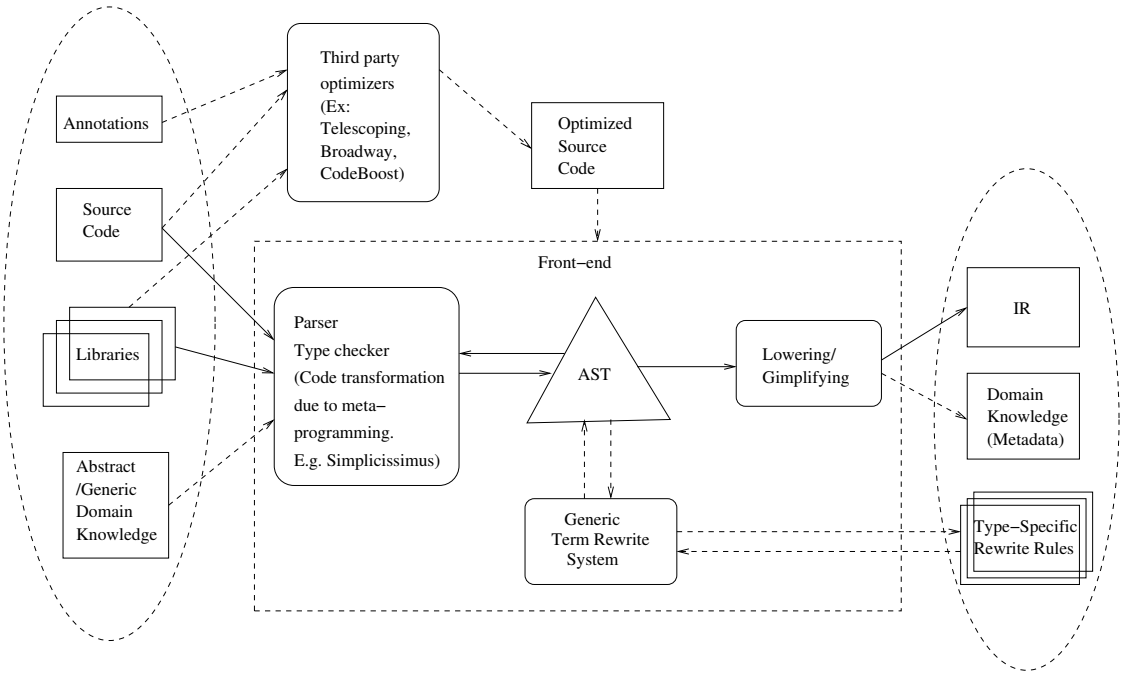

Figure 2.1. Various optimizations techniques applied in the front-end.

optimizers often require users to annotate source code and libraries. This thesis and a few works, such as Simplicissimus [39] and Willcock's Pavilion language [52], utilize the concepts specification [14, 73, 74] to communicate domain knowledge to the compiler. Concepts describe the syntactic and semantic requirements for a class of types, rather than a specific type. Such specification is abstract, generic. One benefit of tying transformations with concepts is that it makes them part of normal routine of generic programming. This assumes a language that supports concepts as a language feature. Indeed this thesis is partially motivated by the concepts features in ConceptC++. It studies how concepts support domain-specific analyses and optimizations. In contrast, Simplicissimus utilizes metaprogramming to emulate concepts, and Willcock investigates how to support concepts in a language which is used for specifying analyses and optimizations.

To utilize the abstract or generic domain knowledge along with source code, this thesis builds a generic rewrite system. This rewrite system, from abstract domain

knowledge, derives type-specific transformation rules and applies them throughout the compilation process. Use of domain knowledge and type-specific rewrites is thus not limited to the front-end. This is an important difference between our work and previous works such as Simplicissimus; Simplicissimus performs optimizations only in the front-end.

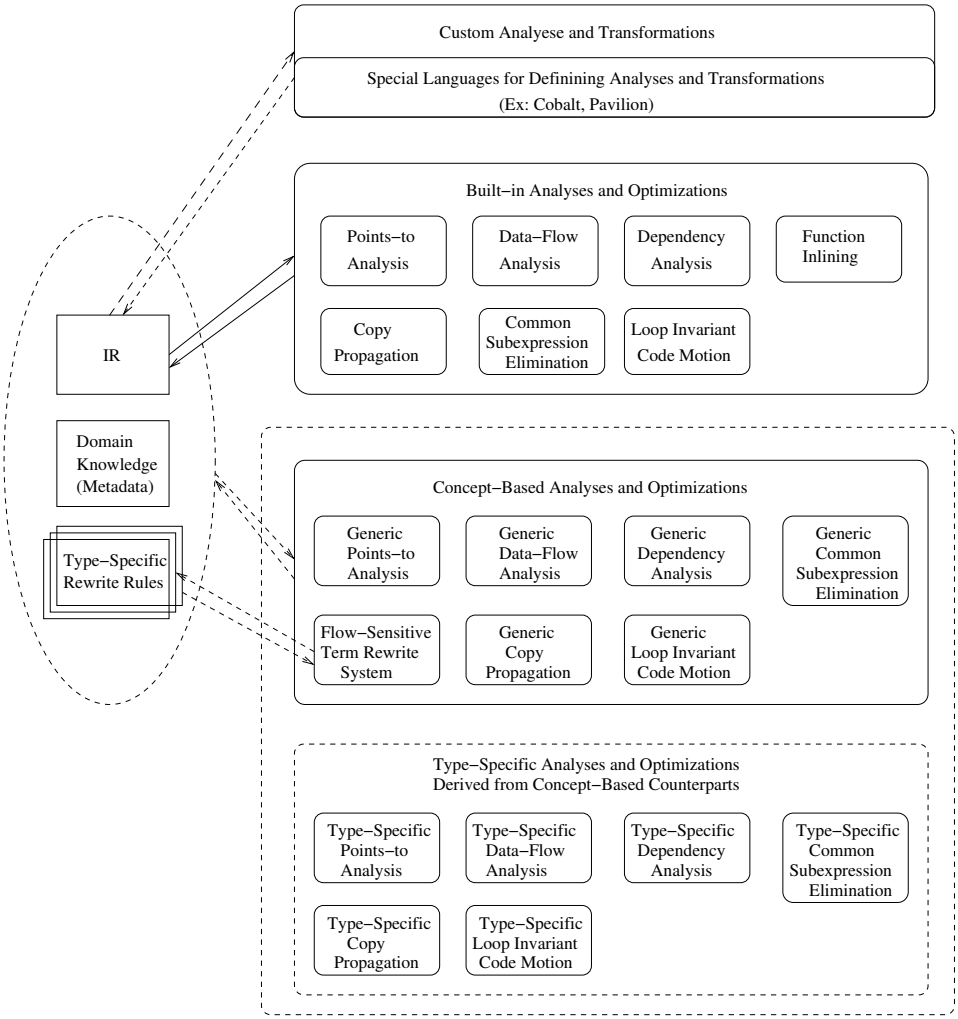

Figure 2.2. Various optimizations techniques used in the middle end.

In the middle-end, as shown in Figure 2.2, typically analyses and optimizations apply to built-in types. To extend these optimizations, one approach is to use declar-

ative languages for defining analyses and transformation. Examples of this approach include Cobalt [51] and Willcock's Pavilion [52]. These works have different focuses from this thesis. They focus on facilitating and designing custom program transformations. Instead, this thesis focuses on usability. It enables the compiler to apply some common transformations to user-defined types and perform user-defined, domain-specific optimizations without little or no efforts from users.

Our approach aims to exploit concepts to define analyses and optimizations that are generic. Generic analyses and optimizations are not defined for specific types but for a class of types, namely concepts. With the available domain knowledge from the front-end, generic analyses and optimizations work as if type-specific analyses and transformations are generated on the fly for optimizing user-defined types and operations. In the long run, generic analyses and transformations may completely cover built-in analyses and optimizations. This thesis, however, keeps both.

Also this thesis performs term rewriting in the middle-end. Together with the term rewriting functionality in the front-end, this thesis presents a full-fledged generic, flow-sensitive term rewrite system. This system requires users to define generic rewrite rules, as part of concepts specification, and attempts then to apply these generic rewrite rules to appropriate concrete types throughout the compilation pipeline. For effective rewriting, it uses data-flow facts that are available from the middle-end analyses.

3. CONCEPT-BASED OPTIMIZATIONS*

The prior chapters describe the potential benefits of high-level optimizations, why they are typically not applied in industrial compilers, and suggest an approach relying on the generic programming paradigm to attain those benefits. The subsequent chapters will detail our contributions in enabling and realizing high-level optimizations in industrial compilers. This chapter gives an overview of the approach and reviews the relevant background.

3.1 Concept-Based Optimization Infrastructure

In generic programming, algorithms whose type parameters are constrained by concepts apply to types that satisfy those concept constraints. This thesis applies the same principle to compiler analyses and optimizations. It studies, when the compiler has knowledge of concepts and of which types model those concepts, how to exploit such knowledge for implementing user-defined, domain-specific optimizations.

Figure 3.1 shows the compiler architecture for taking advantage of concepts for various analyses and optimizations. This architecture is that of a traditional compiler, augmented with an additional component for maintaining the concept and model information. To obtain this information, the front-end parses and analyzes concepts specifications and records the modeling relation between types and concepts. The modeling information is retained during lowering, i.e., the processing of translating high-level programming languages into low-level intermediate representations. The availability of types and concepts at many stages of compilation enables exploiting concepts for various analyses and optimizations.

*Reprinted with permission from "Concept-based optimization", by Xiaolong Tang and Jaakko Järvi. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 97–108, Montreal, Canada, 2007. ©2007 ACM, Inc.

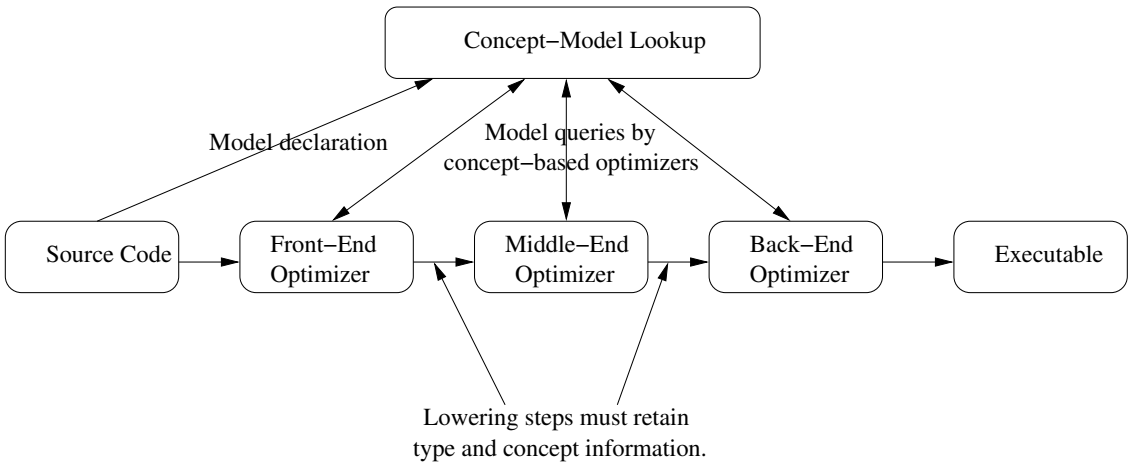

Figure 3.1. Concept-based optimization architecture.

The first possibility of applying concept-based optimizations is at the point of type checking constrained templates. Then, only intra-procedural analysis is possible and the optimization opportunities are thus limited. Our prototype compilers nevertheless apply optimizations at this early phase, immediately after type checking, to simplify further processing. At this point, all necessary information for effecting concept-based transformations is readily available. The constraints on the type parameters of a generic component are used to bind each function and operation call to a particular operation defined in some concept, and thus the transformations that are derived from that concept can be applied. Such transformations are described in Chapter 4.

Many optimization opportunities may only become available after inlining, constant propagation, alias analysis, and other data-flow sensitive information, and it is thus beneficial to apply concept-based optimizations throughout the compilation process. This topic is covered in Chapters 5–7.

3.2 Programmers Perform Generic Optimizations

It is common practice in generic programming that programmers utilize the semantic properties of concepts for ensuring the correctness of algorithms and/or improving code performance. One example is the definition of the advance() algorithm in STL. This algorithm takes as input two parameters, an iterator i and a distance number n, and increases the iterator by the distance number. Depending on the kind of the iterator, there are different optimization opportunities. If the iterator is an input iterator, increasing it by n times can be implemented as while (n−−) ++i. This implementation is not efficient when the iterator is a random access iterator, however. For a random access iterator, the implementation can be as simple as i += n. To accommodate different implementations for a single algorithm, STL utilizes *tag dispatching* [75] and function overloading. In a particular context, an appropriate implementation of an algorithm is selected.

As is clear from the above examples, programmers routinely rely on the semantics specified in concepts to transform code. The key point of this thesis is that a compiler should take advantage of those same semantics for its transformations, and the key contribution of this thesis is to show how this is possible.

3.3 Sources of Concept Taxonomies

Appropriately abstracting type requirements into a hierarchy of concepts is the key to designing such an algorithm as advance(). Figure 3.2 depicts the iterator concepts involved in the advance() algorithm. Concepts in generic libraries typically arise after careful consideration of algorithms and data structures in a particular domain, and have in select domains obtained a standard, or de facto standard, status. Such standard concepts are good candidates for which high-level optimizations should be defined.

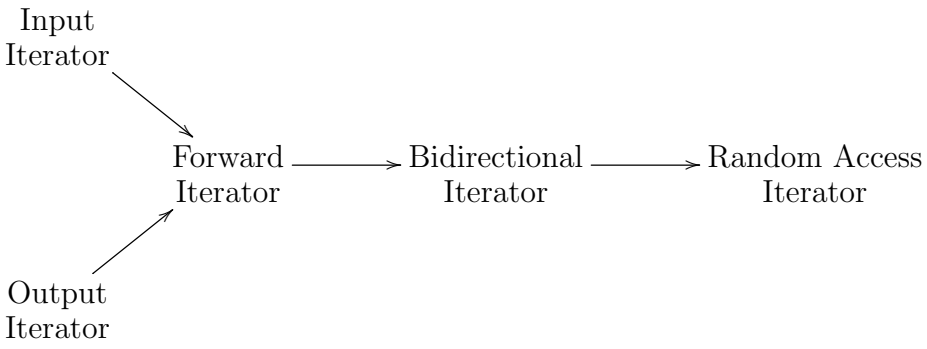

Figure 3.2. Hierarchy for iterator concepts.

Another class of concepts that are of interest for optimizations are mathematical objects. Peter Gottschling describes a concept taxonomy for algebraic structures, and uses it as the basis for building efficient numerical operations in MTL4 [76, 77]. Sibylle Schupp et al. utilize algebraic properties for deriving transformation rules to optimize library function calls [39]. Telescoping languages are an axiomatic system which uses axioms (with algebraic structures) for reasoning about programs [48]. Bagge et al. exploit algebraic properties for optimizing numerical libraries [40]. This thesis, too, uses algebraic concepts as a starting point for supporting domain-specific optimizations. In this thesis, select axioms associated with algebraic concepts are interpreted as transformation rules and they apply to models (concrete algebraic structures) that satisfy these axioms; this way of exploiting concepts agrees with the intended use of axioms in ConceptC++, to be described later in Section 3.5. Chapter 4 and Chapter 5 provide in-depth discussion on this topic.

3.4 Generalizing Built-in Types to Regular Types

Optimizations for built-in types have been well studied. Many user-defined types behave as built-in types. So is it possible to exploit concepts for extending the set of optimizations for built-in types to user-defined types? Dehnert and Stepanov [2] argue about the essential semantics of built-in types which allows reasoning about

the code using built-in types. They introduce the notion of *regular types* to capture the set of types to which the set of optimizations for built-in types may apply. Regular types support default and copy construction, destruction, assignment, and equality comparison in a way that preserves the consistency of these operations. The semantics of regular types specify that the default constructor, copy constructor, and assignment operator leave their target objects in a well-defined state, and that the copy constructor and assignment result in two objects that compare equal [2]. Regular types thus behave, for these operations, as built-in types `int`, `double`, and so on.

In addition to such built-in types, user-defined types whose member variables are of a regular type and who rely on the compiler-generated default and copy constructor, destructor, and assignment operator, as well as an equality operator defined as "memberwise" equality, are regular. Moreover, even many user-defined types that contain pointer fields are regular. For instance, even though the container types of the STL allocate memory from the free store, they are regular. In this category of types, regularity cannot generally be proven by a compiler, but requires a user annotation.

Since regular types generalize built-in types, it is natural to generalize the classes of optimizations for built-int types to regular types. A compiler that is capable of such generalized optimizations requires precise data-flow analysis to understand the behavior of user-defined operations. Traditional program analysis is not aware of the properties of user-defined types and may generate very conservative results in some cases. In contrast, this thesis exploits the concepts knowledge, specifically the properties of regular types, to obtain an affordable, precise, and efficient pointer analysis. This analysis is described in Chapter 6. The available pointer analysis enables many built-in optimizations to be generalized to regular types. Chapter 7 describes some such generalized optimizations.

3.5 Language Support for Concepts: Concepts Feature in ConceptC++

As the piratical realization of our high-level optimizations uses C++ and its "concepts" extensions, we give here a short introduction to ConceptC++.

Concepts in generic programming describe requirements on types. Syntactic requirements in concepts specify what operations must be supported by types to satisfy an interface, and semantic requirements define algebraic laws that must be satisfied by the operations. Commonly requirements are also placed on the complexity of the operations. Concepts are what this thesis uses for the algebraic categorizations of types and for defining the algebraic properties justifying optimizing transformations.

ConceptC++ was designed to extend C++ with complete linguistic support for generic programming, providing a direct representation for concepts in the language. Concept descriptions and their use to constrain type parameters of generic algorithms are immediately useful for enabling modular type checking of templates. This thesis seeks to further take advantage of the concept descriptions as the foundation of an optimization framework.

The central language construct of ConceptC++ is concept. It is used to define sets of requirements on one or more types. Types that satisfy the requirements of a concept *model* that concept. For example, the following (artificially simple) concept requires that the "less than" operator < is defined for objects of type T:

```
concept LessThanComparable<typename T> {
  bool operator<(T, T);
}
```

An explicit declaration, a concept_map, establishes that a particular type (or a parametrized class of types) is a model of a concept. The following two declarations state that the types int and pair<int, string> (serving as a key-value pair here) are models of LessThanComparable:

```
concept_map LessThanComparable<int> { }
```

```
concept_map LessThanComparable<pair<int, string> > {
  bool operator<(pair<int, string> a, pair<int, string> b)
    { return a.first < b.first; }
}
```

These two definitions differ in how LessThanComparable's requirements are satisfied. For int, the built-in < operator for integers suffices. For pair<int, string>, we explicitly define operator< in the body of the concept map (in this case, we order by the first element, i.e., the key). For a concept map to type check, each required operation must either be defined in the concept map's body or in the scope where the concept map is defined.

Though not shown here, the LessThanComparable concept comes with the semantic requirement that the order defined by the < operators is a *strict weak order*. The correctness of many generic algorithms that require their input types to be LessThanComparable (e.g., std::sort) indeed depends on their input types satisfying this semantic requirement. This thesis will rely on such semantic properties to enable optimizations. With the explicit declarations (concept_maps) that types model a concept, programmers also state that the non-syntactic requirements are satisfied.

Concept maps can be templates and can thus adapt entire classes of types at once. For example, the following concept map declares all instances of the standard pair template to be models of LessThanComparable (implementing a lexicographical ordering), assuming the pair's element types are LessThanComparable:

```
template <typename T, typename U>
requires LessThanComparable<T>, LessThanComparable<U>
concept_map LessThanComparable<pair<T, U> > {
  bool operator<(const pair<T, U>& a, const pair<T, U>& b)
  { return a.first < b.first || (!(b.first < a.first) && a.second < b.second); }
}
```

Figure 3.3 shows a simple generic algorithm min_element that uses the LessThanComparable concept as a constraint. Constraints on type parameters are stated in the requires clause. They are enforced at the time of template instantiation and assumed to hold when type checking template bodies. Together, these conditions realize *modular type checking* of templates. The ForwardIterator concept used in the constraints of min_element is also shown in Figure 3.3. This concept provides basic iteration capabilities (capturing the basic notion of a sequence of values). The dereferencing operator $*$ gives the value that an iterator refers to. The ++ operator advances an iterator to the next element. Equality comparison is used to decide when the end of the sequence is reached. Requirements for the == and != operators are not stated directly but obtained through *refinement* of another concept EqualityComparable (not shown). The syntax of refinement is that of class inheritance. The associated type value_type denotes the type of values that the iterator refers to. The requires clause in the concept body places additional constraints on parameters or associated types. Here, value_type must model CopyConstructible, which is a built-in concept, and it has its expected meaning.

Describing the generic min_element algorithm in terms of ForwardIterator allows the algorithm to operate on any sequence of values—whether they are stored in linked lists, hash tables, arrays, or even generated on-the-fly—provided that the value types model the LessThanComparable concept. For example, assuming the concept map for pair<int, string> showed above, the following call (where vec has type vector<pair<int, string>>) satisfies min_element's constraints.

```
pair<int, string> smallest = min_element(vec.begin(), vec.end());
```

3.5.1 Axioms in Concepts

The correct operation of generic algorithms requires much more than simply the existence of certain operations. The type system of ConceptC++ does not deal

```
template <typename Iter>
requires ForwardIterator<Iter>,
         LessThanComparable<Iter::value_type>
Iter min_element(Iter first, Iter last) {
  Iter best = first;
  while (first != last) {
    if (*first < *best) best = first;
    ++first;
  }
  return best;
}

concept ForwardIterator<typename Iter>
    : EqualityComparable<Iter> {
  typename value_type;
  requires CopyConstructible<value_type>;

  value_type& operator*(Iter);
  Iter& operator++(Iter&);
  Iter operator++(Iter&, int);
}
```

Figure 3.3. The min_element generic algorithm and the ForwardIterator concept (simplified from the one in the STL).

with (the generally intractable) semantic requirements attached to concepts, but nevertheless provides a mechanism for their expression. *Axioms* in concept descriptions [78, §7.6] allow programmers to state semantic requirements for concepts in terms of invariants that must hold for types modeling those concepts. ConceptC++ defines no semantics for axioms. They are intended for providing documentation and for conveying information to program manipulation tools. This thesis utilizes axioms for deriving generic optimization rules.

This section uses the algebraic structure of *Monoid* to illustrate axioms in concepts. The *Monoid* concept requires two operations, the constant identity function and a binary operation. It can be defined in ConceptC++ as follows:

```
concept Monoid<typename Op, typename T> {
```

```
  T identity_element(Op, T);
};
```

Note that this concept is parametrized in both the carrier type T and the binary monoid operation, as these are what uniquely characterize the structure.

The above definition does not fully capture what it means to be a *Monoid*, since the concept lacks the semantic requirements *identity axiom* and *associativity axiom*. The following code uses ConceptC++'s axiom construct to add the first of these requirements into the Monoid concept:

```
concept Monoid<typename Op, typename T> {
  T identity_element(Op, T);
  axiom Identity(Op op, T x) {
    op(x, identity_element(op)) == x;
    op(identity_element(op), x) == x;
  }
};
```

The syntax of axioms is that of functions but with the keyword axiom as the placeholder for function return types. Here, Identity names our identity axiom, and its body contains the equations describing the invariants comprising the axiom: the right identity law and left identity law. The definition of Monoid is still incomplete in other ways; Figure 4.3 shows the complete definition.

Axioms allow the specification of equations and inequalities. Equations are of interest for this work as they can give rise to transformation rules. For instance, the two identity laws permit to replace expressions with their simplified versions (e.g. x in place of op(x, identity_element(op))), resulting in performance improvement. As the equations in axioms hold for all types that model a concept, transformation rules arising from the equations potentially apply widely. For example, the transformation rules in Figure 4.1 are instances of monoid's identity laws with the tuples <int, +, 0> and <int, $*$, 1> as monoids. Many user-defined types satisfy the same laws and can

be declared as models of Monoid. Examples include a multiplicative monoid for an unbounded precision integer <bignum, $*$, bignum(1)> or for C++'s standard string class with concatenation <string, +, string()>.

Note that this work does not attempt to formally verify that axioms are respected. This is left to the programmer to guarantee. Examples of relying on programmers to provide a certain semantics for user-defined types, and relying on that in optimizations, are common. For example, C++ compilers can elide copy constructor invocations, even if the constructors have side effects [79][§12.8], and Haskell compilers are allowed to assume that so called *Monad laws* hold for types that are instances of the Monad type class [80, 81].

4. AXIOM-BASED OPTIMIZING TRANSFORMATIONS*

Consider traditional optimizing transformations based on the properties of the built-in operations and types hard-coded into a compiler. Often transformation rules for different built-in types are essentially isomorphic, yet the transformations are implemented as separate rules. For example, Figure 4.1 shows the simplification rules $x + 0 \longrightarrow x$ and $x * 1 \longrightarrow x$ for the built-in integer type in GCC [82]. As seen, these rules are defined in two distinct cases. However, both transformations are justified by the same algebraic property, *right identity*, that holds in all *monoids*. Here, in the first transformation, integer can be viewed as a monoid with addition as the binary operation, and 0 as the identity element. In the second transformation, multiplication is the binary operation, and 1 the identity element.

```
tree fold_binary (enum tree_code code, tree type,
                  tree op0, tree op1) {
  switch (code) {
    case PLUS_EXPR:
      if (! FLOAT_TYPE_P (type)) {
        if (integer_zerop (arg1))
          return non_lvalue (fold_convert (type, arg0));
        ...
    case MULT_EXPR:
      if (! FLOAT_TYPE_P (type)) {
        if (integer_onep (arg1))
          return non_lvalue (fold_convert (type, arg0));
        ...
```

Figure 4.1. Excerpt from the simplifier in the GNU compiler.

As described in Chapter 3, types can be categorized into concepts, and transformations rules defined as *axioms* of those concepts. Where two isomorphic trans-

*Reprinted with permission from "Concept-based optimization", by Xiaolong Tang and Jaakko Järvi. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 97–108, Montreal, Canada, 2007. ©2007 ACM, Inc.

formation rules are today repetitively implemented as separate cases, in such an approach they become two different instances of the same generic rule. Moreover, the generic rule applies in all cases where a type with its operations models the appropriate concepts. The right identity rule above, e.g., applies to a user-defined string data type with string concatenation as the binary operation and the empty string as the identity element of a monoid. When axioms in concepts are interpreted as generic transformation rules, their application to particular types are justified by the concept maps that establish that these particular types model the concepts defining these axioms.

The contribution of this chapter is a framework for "axiom-based optimizing transformations" for ConceptC++, implemented on top of the ConceptGCC compiler [12]. Specifically, this chapter examines interpreting axioms in concepts as transformation rules and identifies a set of concepts relevant for compiler transformations. It describes the design and implementation of a prototype concept-based simplifier that recognizes these concepts and applies transformation rules derived from them generically to all types that are models of the concepts.

This chapter draws ideas from the Simplicissimus framework [39, 83], which also used concepts for describing and implementing generic compiler transformations. The work on Simplicissimus preceded the built-in concepts extensions of C++. Consequently, properties on operations justifying transformations were expressed with C++ "traits" [84] and other advanced template techniques, instead of using explicit concept descriptions for this purpose. Simplicissimus relied heavily on so called "C++ template meta-programming" to effect its transformations.

4.1 Axiom-Based Optimizations

Algebraic identity optimizations [85–87] have been well established in traditional compilers. The goal of this chapter is to generalize them from special rules for built-in types like int and double, or for particular vector or matrix classes, to general rules for

abstract algebraic categories, i.e., concepts. To accomplish this, it is necessary to first define the representations for the algebraic concepts like SemiGroup, Monoid, Group, Ring, and Field [88] in ConceptC++. These concepts capture the essential algebraic properties of the above mentioned numerical types, and the algebraic laws of the concepts are candidates for transformation rules.

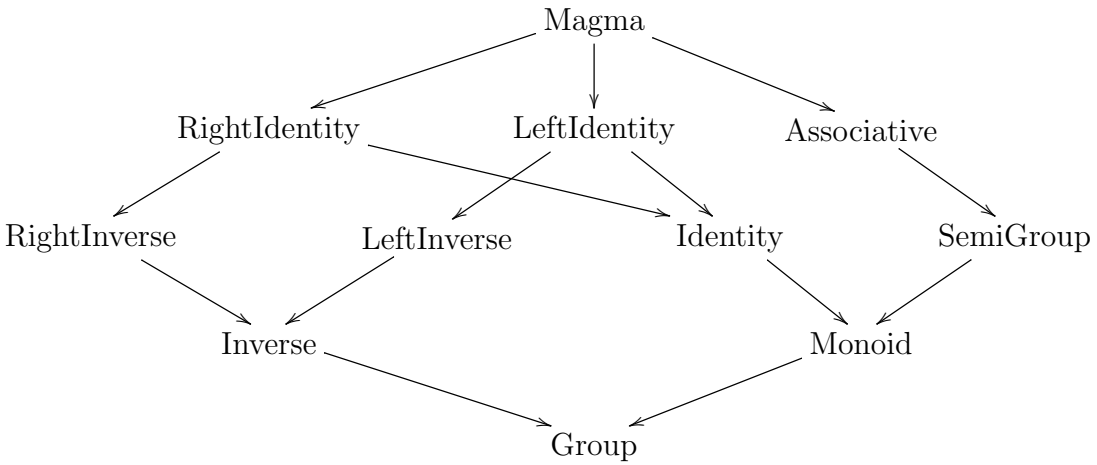

Figure 4.2. Hierarchy of algebraic concepts

Figure 4.2 depicts a taxonomy of the algebraic concepts studied in this chapter. The lines between concepts represent refinement. Some of the concepts introduce new operations, some merely add new axioms. Magma represents a set with a binary operation; RightIdentity, LeftIdentity, and Associative refine Magma, introducing one operation identity(), and observing the left identity axiom $op(identity(op, x), x) == x$, the right identity axiom $op(x, identity(op, x)) == x$, and the associative axiom $op(op(x, y), z) == op(x, op(y, z))$ respectively; RightInverse and LeftInverse are refinements of RightIdentity and LeftIdentity, introducing the new operation inverse(), and observing the left and right inverse axioms $op(inverse(op, x), x) == identity(op, x)$ and, respectively, $op(x, inverse(op, x)) == identity(op, x)$. The Identity concept refines both the LeftIdentity and RightIdentity concepts; SemiGroup refines Associative; the Inverse concept comprises of requirements from both RightInverse and LeftInverse;

the Monoid concept augments SemiGroup with Identity requirements; and finally Group refines Monoid and Inverse.

The axioms in these concepts serve as transformation rules, interpreted from left to right. The rules are applicable to all types that are established to be models of these concepts, and are used to simplify expressions. Figure 4.3 shows the definitions of the Magma, Associative, SemiGroup, and Monoid concepts.

To illustrate the applicability of the generic transformation rules, Table 4.1 summarizes some of the types and operations that are models of Monoid and can take advantage of the identity rules.

```
concept Magma<typename Op, typename T> {
  requires std::Callable<Op, T, T>;
  requires std::SameType<
    std::Callable2<Op, T, T>::result_type, T&>;
}
concept Associative<typename Op, typename T>
    : Magma<Op, T> {
  axiom Associativity(Op op, T x, T y, T z) {
    op(x, op(y, z)) == op(op(x, y), z);
  }
}
concept SemiGroup<typename Op, typename T>
    : Associative<Op, T> {}
concept Monoid<typename Op, typename T>
    : SemiGroup <Op, T> {
  T identity_element(Op);
  axiom Identity(Op op, T x) {
    op(x, identity_element(op)) == x;
    op(identity_element(op), x) == x;
  }
};
```

Figure 4.3. An excerpt of definitions of algebraic concepts.

The taxonomy in Figure 4.2 was obtained as a result of analyzing the existing algebraic simplification rules in an industrial strength C++ compiler [82]. Results of the analysis suggests extending the taxonomy further. Simplifications with more than two operators are common. E.g., GCC takes advantage of the distributivity laws $A*C+B*C == (A+B)*C$ and $X*C_1+X*C_2 == X*(C_1+C_2)$. More sophisticated concepts, such as Ring and BooleanAlgebra, capture many common optimization cases with more than two operators. This thesis does not cover those concepts; it focuses on demonstrating the feasibility and benefits of exploiting axioms and concepts for optimizations, but does not describe a full-fledged generic optimizer.

Data type	Operation	Identity Element
`int`	`+`	`0`
`int`	`*`	`1`
`set<T>`	`union`	`set<T>()`
`bool`	`&&`	`true`
`string`	`+`	`string()`
`bignum`	`+`	`bignum(0)`
`matrix(m,n)`	`+`	`zero_matrix(m,n)`

Table 4.1
Several models of Monoid

4.2 Framework and Implementation

The concept-based optimizer effecting the optimizations described here is implemented by extending the ConceptGCC compiler [12]. This extended compiler is available in ConceptGCC's public subversion repository [89]. The concept-based optimizer takes effect if the compiler is given the `-fconcept-simplify` option.

Axiom-based optimizations span across the whole compilation process, from the front-end to the back-end. Figure 4.4 depicts how the concept-based optimizer integrates to the existing structure of ConceptGCC. The optimizer interacts with three

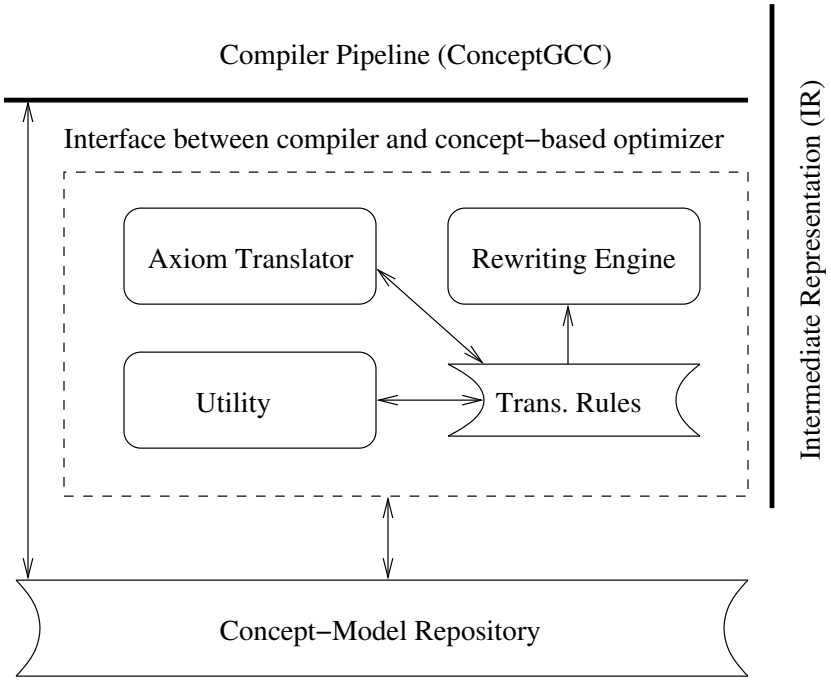

Figure 4.4. Framework of the concept-based optimizer for exploiting axioms for optimizations.

entities: it provides a set of hook functions for the compiler pipeline to initiate operations for concept-based optimizations, such as generating transformation rules, pattern matching over target expressions, and conducting rewriting operations; it accesses the concept-model repository for axioms and modeling relationships when generating transformation rules; and it analyzes internal representations of the program at various stages of compilation and applies the axiom-based transformation rules. As indicated in Figure 4.4, the optimizer itself consists of three sub-modules: the axiom translator, the rewriting engine, and a sub-module of utility functionality. The utility sub-module comprises of basic operations like initialization and resource recycling. The other two sub-modules are discussed below.

4.2.1 Optimizer Interface

The concept-based optimizer executes axiom-based transformations at two stages of compilation: first in the front-end, second in the middle-end. In the front-end, it optimizes constrained template functions. The front-end consults the axiom translator to prepare transformation rules before parsing the body of constrained template functions, and, after type-checking them, initiates the rewriting engine to conduct the pattern matching and transforming operations over statements or expressions within the function body.

Whereas the front-end optimization processing is interleaved with parsing and type checking of templates, the middle-end processing occurs in a separate pass devoted for axiom-based optimization. This pass is registered into GCC's existing optimization framework whose "pass manager" manages optimization processes, applying a sequence of optimizer callbacks to each function node in a pre-defined order.

In the middle-end, the optimizer targets non-generic code. Thus, the optimizer has to instantiate generic transformation rules for particular types guided by the concept map definitions. For instance, to apply generic algebraic simplification rules to built-in types, it generates transformation rules for specific types based on program-wide concept maps, as discussed in Section 4.4.2.

Extending the axiom-based optimizations to the middle-end is described in the following chapter. The goal is to take advantage of information regarding aliasing, constant and copy propagation, and of inlining.

4.2.2 Rule Translator

The task of the rule translator is to generate effective transformation rules. This consists of two subtasks. First, the rule translator interprets axioms defined in concepts and generates transformation rules to be applied, in the front-end, directly to uninstantiated templates. These rules transform bodies of function templates

independently of any of their instantiations. Second, the rule translator lowers already established generic rules to be used in the middle-end optimization pass. At that point, templates have already been instantiated, and the lowered rules apply to operations on concrete types. Therefore, the concept-based optimizer retains the information of what concepts those concrete types model, and what concepts were used as constraints of the function templates from which the code to be optimized was generated.

Transformation rules generated by the rule translator are matched against the AST or other internal representations of the program. The representation of the patterns that describe the left-hand side of the transformation rules is thus important for efficient matching of rules. In the front-end, the first-child next-sibling representation of a tree [90, 91] is chosen as the pattern representation of transformation rules.

As an example, Figure 4.5 demonstrates the representation of the left-hand side of the right identity law op(x, identity_element(op)) defined in the Monoid concept. To find a match, the optimizer recursively traverses the program's AST to match the patterns against the AST nodes.

4.2.3 Rewriting Engine

Once the rule translator has generated the transformation rules, the rewriting engine pattern matches each expression in a function to be optimized against the patterns of those rules that are in effect in the function. When a redex is identified, it is replaced with its contraction that results from applying the substitution (that validates the redex as an instance of a pattern) to the right-hand side of the corresponding transformation rule. The substitution is obtained with normal unification mechanisms.

```
[1      e06d20      pck_fun_cell      call_expr      ...]
[1.1    42d1b000    pck_const_cell    integer_cst    ...]
[1.2    e03e70      pck_var_cell      var_decl       ...]
[1.3    e062a0      pck_var_cell      var_decl       ...]
[1.4    e062a0      pck_fun_cell      call_expr      ...]
[1.4.1  42d1b000    pck_const_cell    integer_cst    ...]
[1.4.2  e0ccb0      pck_code_cell     function_decl  ...]
[1.4.3  e03e70      pck_var_cell      var_decl       ...]
```

Figure 4.5. The internal representation of the right identity law from Monoid as generated by the rule translator. Each row stands for a tree node, where position, kind, cell category and original tree code are shown from columns one to four. For example, the root node (1) is a call expression and corresponds to the function invocation op(); node (1.1) is the operand number of the function; and nodes (1.2), (1.3), and (1.4) correspond to the arguments of the function; the self object; x, and identity_element(op), respectively.

The rewriting engine tries to match transformation rules in a pre-defined order a fixed number of times. It uses the leftmost-outermost reduction strategy: for a particular pattern, it sequentially selects *redexes* from left to right, and the outermost *redex* presents a higher reduction priority than the innermost one.

The rule generation for effective pattern matching and application of the concept-based optimization rules in the middle-end optimizer is somewhat challenging. The GCC compiler lowers non-template code and instantiated template code to so called GIMPLE form [92] immediately after parsing and type-checking. GIMPLE provides a language-independent representation suitable for optimizations—most optimizations are performed after lowering ASTs into the GIMPLE form. GIMPLE retains much of the structure, lexical scopes and control constructs, of the parse tree: functions are represented as trees and loops as containers.

GIMPLE breaks expressions into 3-address form, using temporary variables to hold intermediate values. Considering that the compiler directly compiles non-generic functions into GIMPLE trees, it is necessary to represent optimization rules

in a manner that is compatible with this form for pattern matching and for applying transformations. A suitable representation is described in the next chapter.

4.2.4 Concept-Model Repository

The concept-model repository is accessible in all phases of compilation where concept-based rules are applied. The repository holds information of every concept in the program and the information of which types are models of which concepts. This information is obtained during parsing and semantic analysis in the front-end of the compiler.

Much of the necessary information is already present in existing constructs of the compiler. E.g., the modeling relationship is accessible by looking up the "specializations" of the concept with existing mechanisms of GCC (concepts do not really have specializations, but as concepts are internally represented as class templates in ConceptGCC [12], internally they do), and the models associated with a constrained template are reachable by going through the requires clause of the template. The concept-model repository provides a central access mechanism to all information necessary for applying the concept-based transformations.

4.3 Example

This section shows an example of how a high-level optimization rule is communicated to the compiler using concepts and axioms. We also describe the effect of the optimization on a simple program, thus confirming that the compiler successfully recognizes the rule. The example uses the Monoid concept and demonstrate how its generic identity laws enable optimizations for user-defined types, in this case operations on the Matrix Template Library's matrix types; matrices with addition as the binary operation and the zero matrix as the identity element form a monoid. Thus, as long as it is established that MTL's matrix with its addition operation model the

Monoid concept, identity transformations of Monoids become applicable to matrices. Figure 4.6 declares a specific matrix type from MTL and the operation plus<Matrix> as a Monoid. For technical reasons, the concept map declaration includes the second argument (of type Matrix) in the identity_element function. Also, due to the way in which the Monoid concept is expressed, parametrized by both the carrier set and the binary operation, MTL's add function is wrapped into an instance of the plus function object—a function pointer cannot directly serve as a type parameter.

```
typedef mtl::matrix<double, mtl::rectangle<>,
  mtl::array<mtl::dense<> >, mtl::row_major>::type Matrix;
concept_map Monoid<mtl::plus<Matrix>, Matrix>{
  Matrix identity_element(mtl::plus<Matrix> op, Matrix x) { return mtl::zero(x); }
}
```

Figure 4.6. Declaration that specifies an instance of mtl::matrix and the operation plus<matrix> as a monoid.

The concept map in Figure 4.6 is the only piece of code specific to MTL matrices to enable the identity law optimizations for matrix addition. Figure 4.7 demonstrates an application of the optimization. Because the template parameters of the mtl_opt_test function are constrained by the concept Monoid, the axioms defined in Monoid are effective in the scope of that function. As a result, the expression op(t, identity_element(op, t)) is optimized to be t, which is later optimized away due to dead code elimination. In main(), M and N come from command line parameters.

Figure 4.8 shows the results (opt−add denotes the optimized executable with −fconcept−simplify flag and add the unoptimized version). The execution time without optimization comprises of the time to initialize the matrices and perform a matrix addition, and it thus grows linearly with the size of the problem (number of elements in the matrices). With concept-based optimization the execution time consists only of

```
#include <concepts>
#include "algconcept.hpp"
#include "mtl/mtl.h"

namespace mtl{
  template <class Matrix>
  struct plus : std::binary_function<Matrix, Matrix, Matrix> {
        Matrix operator() (Matrix a, Matrix b) { add(a, b); return a; }
      };

  template <class Matrix>
  Matrix zero(Matrix A) { zero_matrix(A); return A; }
}

template <typename Op, typename T>
requires Monoid<Op, T>
void mtl_opt_test (Op op, T t) { op(t, identity_element(op, t)); }

int main(int argc, char* argv[]) {
  ...
  Matrix x(M, N);
  mtl_opt_test (mtl::plus<Matrix>(), x);
  ...
}
```

Figure 4.7. Code that contains the identity law optimization opportunity for matrix addition.

the initialization of the matrices, and stays thus 5-6 times lower. With optimization, the size of the executable also shrinks.

The example demonstrates that with little or no additional effort for the programmer, the optimizer can perform high-level optimizations on operations of user-defined types. In contrast, the traditional approach of lowering the high-level operation into operations on built-in types, and then attempting to optimize, cannot recover the lost semantic information, resulting in significantly less efficient code.

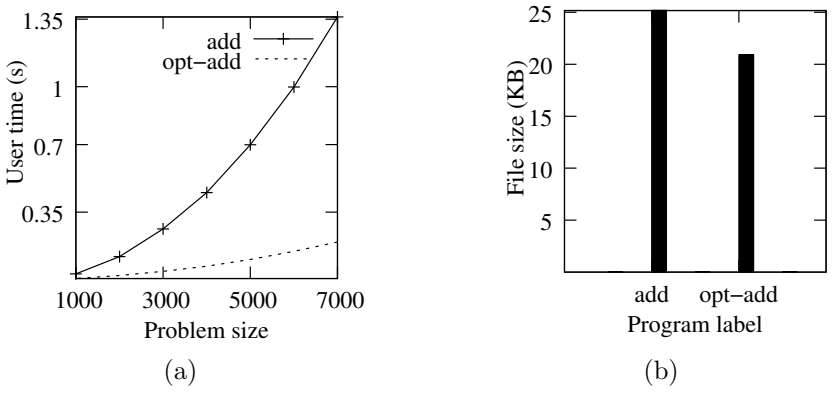

Figure 4.8. Executable sizes (a) and execution times (b) of the test program with and without concept-based optimization. The execution times were measured with varying matrix sizes; the unit of x-axis is M, where the size of the matrix is $M \times M$.

4.4 Discussion

Not all concepts and axioms should be exploited for optimizations. Therefore, caution is needed for ensuring the appropriate use of concepts and axioms for optimizations. Regarding this concern, the feature of auto concepts; the scope of concept-based optimizations; the properties of the rewrite system which is formed by the set of transformation rules derived from axioms; and the strategy for applying these transformation rules must be considered.

4.4.1 Auto Concepts

The language construct concept_map serves as a mapping between types and concepts, establishing the relation that a type models a concept. This explicit declaration by the programmer establishes that a type satisfies (or types satisfy) the semantic requirements of a concept—violations of syntactic requirements are caught by the compiler. Explicit concept maps thus justify that certain generic algorithms, and *generic optimizations*, apply to particular data types.

ConceptC++ supports a special form of concepts: auto concepts. For these concepts, no explicit concept map declaration is needed to establish a models relation. All that is needed is that the syntactic requirements are satisfied, that is, all required operators and functions are defined. Regarding the implementation of our concept-based optimizer, this poses no difficulty for us. Even in the case of auto concepts, structures corresponding to an explicit concept map are created internally, and can be used for realizing concept-based transformations. Regarding the correctness of transformations, however, auto concepts raise a concern; when mere structural conformance (presence of certain user-defined operators and functions) suffices to enable certain transformations, special care is needed to guarantee that the semantics of those operations are such that correctness is preserved. It is observed that auto concepts in the "conceptualized" standard library are typically concepts with a single operation requirement and no algebraic laws attached to the operation—CopyConstructible, Assignable, and Callable are typical examples. Such concepts do not introduce optimization opportunities. Furthermore, separate annotations can be used to select the set of concepts that the compiler considers for optimization transformations; auto concepts can be excluded from this set.

4.4.2 Scope of Concept-Based Optimizations

A concept map is only in effect in a context where a type is constrained with the corresponding concept. For example, consider the concept map Section 3.5 that made all std::pairs models of LessThanComparable. That concept map defines operator< to provide a lexicographical ordering for pairs. The operator, however, is only in scope in instantiations of generic classes or functions where a pair instance is bound to a type parameter constrained by LessThanComparable. The concept map has no effect outside generic functions. This design is justified—concept maps are a new layer on top of the existing overloading mechanism of C++ and concept maps are geared for adapting interfaces. Concept maps define views that are only active when requested,

which is a desirable trait for adaptation and library composition [93]. However, for enabling high-level optimizations, this characteristic of concept maps is a limitation—ConceptC++ provides no mechanism for establishing a "type models concept"-relation (which is necessary for activating concept-based optimizations) that would apply outside generic definitions.

If we wish to replace most type-specific simplification rules in a compiler with instances of generic algebraic rules, it is necessary that concept-based optimization can be extended to non-generic code. One strategy is making the compiler aware of a class of concepts and a set of models declarations, e.g., that int with operator + and constant 0 is a model of Monoid, and that int with operator $*$ and constant 1 is a model of Monoid. Compiling the entire program assuming that such a set of concepts and concept maps are in effect enables the generic rules in non-generic code—and the same level of optimization as with non-generic rules but with fewer special cases.

Furthermore, the approach of program-wide concept maps makes it relatively easy to extend the set of built-in optimizations to apply more widely, say, to types and operations defined in the standard library. For example, the standard string class, with concatenation operation + and the empty string as the identity element are a model of the Monoid concept, from which we get the optimization rules x + std::string("") == x and std::string("") + x == x.

4.4.3 Properties of Rewriting System

The rewriting system emerging as a combination of an unrestricted set of concept-based transformation rules is unlikely to exhibit nice properties, such as termination and confluence (termination guarantees that no infinite rewriting sequence exists, whereas confluence asserts that the rewriting order does not matter for the final result [94]). These properties seldom hold for optimizations in practical compilers either [95]. Consequently, transformations are typically tried and applied in a set order, a bounded number of times, rather than applied repeatedly until a fixed

point is reached, or attempted to be solved as one set of equations. The benefits of this "engineering" approach are that the concept-based optimization framework makes it relatively easy to implement new transformations, and more importantly, enables existing built-in optimizations to apply to user-defined types—the set of rules to include and the strategy of applying them is still built into the compiler. An interesting direction of future research is to investigate the expression of such optimization strategies as libraries.

4.5 Conclusions

Concepts describe a set of properties of a class of types. Programmers state, using concept maps, which types possess those properties and belong to particular classes of types. Concepts can be utilized in optimization at two levels. First, it is possible to express transformations for sets of types collectively and have the transformations apply to an open-ended collection of types, extensible by the programmer. Second, optimization rules can be directly derived from the description of concepts—in particular, the axioms in concepts can be interpreted as rewrite rules. The focus of this chapter was on the latter level. The following chapters focus on the former level and move towards more general ways of exploring concepts for optimizations.

The concept-based optimization framework in this chapter supports the definition of arbitrary transformation rules as axioms of a concept. The transformation rules apply intra-procedurally in contexts that are explicitly constrained by the relevant concepts. The next chapter extends the concept-based optimization rules to be applied also at later phases of compilation to uncover substantially more optimization opportunities, e.g., after inlining, constant propagation, and utilizing the results of data-flow analysis. Next steps after that include establishing the core set of concepts and axioms that result in widely applicable effective optimizations for a concept-based optimizing compiler. One measure of success this work strives for is whether

the concept-based optimization framework can largely replace type-specific built-in algebraic simplification rules.

The approach of axiom-based optimization utilizes axioms for specifying rewrite rules. These rewrite rules are generic,and applying these generic rewrite rules to non-generic code is challenging. To be concrete, consider the example of applying the identity axioms in monoid to the std::string class.

Recall the Monoid concept. Figure 5.1 shows a definition for this concept, which is simplified from a proposed taxonomy [96] of *algebraic concepts* for ConceptC++: The

```
concept Monoid<typename Op, typename T> : SemiGroup<Op, T> {
  T identity_element(Op, T); // identity element
  axiom Identity(Op op, T x) {
    op(x, identity_element(op, x)) == x; // right identity law
    op(identity_element(op, x), x) == x; // left identity law
  }
}
```

Figure 5.1. A simplified definition for the Monoid concept.

two type parameters Op and T represent an operation and a set, respectively. The identity_element() function corresponds to the identity element in a monoid, and the axiom specification defines monoid's identity laws. By refining the concept SemiGroup, the Monoid concept inherits all requirements defined in SemiGroup, notably that Op is an associative binary operation.

The Identity axiom expresses two generic transformation rules, one for the left identity and the other for the right identity. These transformation rules are defined at a high level of abstraction—the op operation and the identity_element() function which are involved in these transformation rules are not in the context of non-generic code.

*Reprinted with permission from "Generic flow-sensitive optimizing transformations in C++ with concepts", by Xiaolong Tang and Jaakko Järvi. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2111–2118, Sierre, Switzerland, 2010. ©2010 ACM, Inc.

The two identity rules shall be lowered to ones which are suitable for optimizing non-generic code.

A concept map for the Monoid concept justifies applying these generic transformation rules to a particular type. At the same time, this concept map provides the clue about how to translate these transformation rules to optimize the operations for this particular type in non-generic code. For example, Figure 5.2, a concept map declaration, states that the binary operation of the model is the string concatenation, which is not shown, wrapped into the *function object* plus<string>. It also states that the identity element is std::string(""), as specified in the the implementation of the identity_element function.

```
concept_map Monoid<plus<std::string>, std::string> {
  std::string identity_element (plus<std::string> op, std::string x) {
    return std::string("");
  }
}
```

Figure 5.2. The declaration for specifying that the std::string class and the string concatenation forms a model of the Monoid concept.

Given the above model, the goal is to generate these two concrete transformation rules for the string class:

$$x + \text{string}("") \rightarrow x \tag{5.1}$$

$$\text{string}("") + x \rightarrow x \tag{5.2}$$

This chapter describes an approach that accomplishes the translation from generic transformation rules to concrete transformation rules. This approach leverages the existing processing functionality of the compiler and does not affect the normal pipeline of the compiler.

Our approach also addresses a limitation with simple term rewriting. In simple rule-based rewriting, pattern matching is often confined to a single expression or statement; it fails to "see across a semicolon." For example, consider the rewrite rule $f(x, g(y)) \rightarrow h(x, y)$. Its left hand side (LHS) matches an expression like `f(a, g(b))`, but a seemingly inconsequentially transforming (adding a temporary variable) the expression into {`t = g(b); f(a, t)`} may hide the rewriting opportunity. Robison argues that optimizers should be robust for the above kinds of transformations [97].

For robust high-level optimizations, the approach of this chapter utilizes a flexible representation for rewrite rules. It transforms rewrite rules into *conditional* rewrite rules, so that data-flow analysis can be exploited for more flexible optimizations. For instance, the above rewrite rule is transformed into

```
f(x, t) | { def(t) = g(y) } → h(x, y)
```
(5.3)

which reads as the normal rewrite rule `f(x, t)` → `h(x, y)` with an extra condition `def(t) = g(y)` on the LHS, insisting that t is defined to be the result of `g(y)`.

With the flexible representation for rewrite rules, the approach extends axiom-based optimizations into the middle-end of the compiler, and combines the compiler's existing analyses and transformations with axiom-based optimizations. In particular, this approach proposes a strategy for combining function inlining with axiom-based optimizations for addressing the phase-ordering problem [98, 99].

Specifically, this chapter makes the following contributions.

- It presents an approach for exploiting concepts and axioms for supporting domain-specific optimizations in the middle-end of the compiler. It shows how to utilize the compiler's middle end for high-level rewriting, including instantiating generic rewrite rules to produce type-specific rules to be applied by the middle end, and controlling function inlining to avoid loss of rewriting opportunities.

- It provides a prototype for the approach. The prototype is implemented as an extension to ConceptGCC [12]. The experiments with this prototype show that the approach effectively reduces abstraction penalties without a significant increase in compilation times.

The remaining of this chapter is organized as follows. Section 5.1 details the steps of translating generic rewrite rules into type-specific rewrite rules, describes the combination of term rewriting with function inlining, and presents an approach that exploits concepts and axioms for algebraic simplifications for user-defined operations. Section 5.2 evaluates the approach. Finally Section 5.3 concludes this chapter.

5.1 Generic and Flow-Sensitive Rewriting

The axiom feature of concepts offers a means to specify generic rewrite rules, applicable to all types that model a particular concept. The syntax of axioms can only express an equivalence between two expressions, not a rewrite rule.. Therefore, additional conventions or annotations (e.g., with the help of C++11's *attribute* mechanism) are necessary to specify a direction of applying a rewrite. In our prototype, we interpret each equation in an axiom as a left-to-right rewrite rule. Consider again the Monoid concept in Figure 5.1. The two equations in the body of the Identity axiom result in the generic *left identity rule* $\mathcal{R}4$ and the generic *right identity rule* $\mathcal{R}5$.

$$\mathsf{op}\ (\mathsf{identity\_element}\ (\mathsf{op}, \mathsf{x}), \mathsf{x}) \rightarrow \mathsf{x} \qquad (\mathcal{R}4)$$

$$\mathsf{op}\ (\mathsf{x}, \mathsf{identity\_element}\ (\mathsf{op}, \mathsf{x})) \rightarrow \mathsf{x} \qquad (\mathcal{R}5)$$

These generic rules express the common pattern of many type-specific rules. Table 5.1 lists several such "non-generic" rewrite rules, all instances of the generic $\mathcal{R}5$ rule. This table is similar to Table 4.1 but the rule instances have been added. In our approach, the instances are not hard-coded into the compiler, but instead generated from the generic rule of Monoid for all models of that concept. For example, the

concept map in Figure 5.2 justifies generating the left and right identity rules for the string class.

Type	Op	Identity	Rewrite Rule
int	+	0	i + 0 -> i
int	*	1	i * 1 -> i
set<T>	union	set<T>()	union(s, set<T>()) -> s
bool	&&	true	b && true -> b
string	+	string()	s + string() -> s
bigint	+	bigint(0)	i + bigint(0) -> i
matrix(m, n)	+	zero_matrix(m, n)	r + zero_matrix(m, n) -> r

Table 5.1

Several models of Monoid along with their corresponding specific right identity rules. The triple of the values in the first three columns describes a particular monoid, and the fourth column shows the instance of the right identity rule in that monoid.

5.1.1 Conditional Rewrite Rules

Directly mapping axioms to rewrite rules leads to rules with rigid patterns, unlikely to unveil many optimization opportunities. E.g., the rewrite rules in Table 5.1 directly match only to the abstract syntax tree (AST) of a single expression where the right-hand side (RHS) operand is the identity element; an operand equivalent to but not literally the same as the identity element would prevent the rule from being applied. Figure 5.3 illustrates the limitation using the rule $x + \text{string}(\text{""}) \rightarrow x$ as an example: the code fragment (a) can be transformed because one of its expressions contains an exact match against this rule's LHS; the fragments (b) and (c) both have expressions that evaluate to a value equivalent to the rule's LHS, but do not constitute a match.

```
string x ("text");
string z = x + string ("");
```

(a)

```
string x ("text");
string y ("");
string z = x + y;
```

(b)

```
string add (string a, string b) {
  return a + b;
}

void main () {
  string x("text"), y("");
  add (x, y);
}
```

(c)

Figure 5.3. Code fragments that contain equivalent expressions to the LHS of the rewrite rule x + string("") → x.

To improve the robustness of the transformations, data-flow facts that are expressed in rewrite rules are separated from rule patterns. This separation give rises to conditional rewrite rules. For example, the conditional rule for the right identity rule for string is defined as:

$$\mathsf{x} + \mathsf{y} \mid \{\ \mathsf{def(y)} = \mathsf{string("")}\ \} \rightarrow \mathsf{x} \qquad (\mathcal{R}6)$$

The transformation of a generic rewrite rule to a conditional rewrite rule is mechanical, and is explained in 5.1.2.

Applying the above rule requires the pattern matching to recognize the operator+() overload for string, followed by ensuring, based on data-flow information, that the definition for the operator's second argument is equivalent with the result of the constructor call string(""). This strategy of rule application allows for rewriting the fragment (b) and (c) in Figure 5.3, as long as the compiler's analyses are powerful enough to recognize the equivalences. Section 5.1.2 discusses how to utilize the compiler's existing analyses and extend them towards this goal.

5.1.2 Processing Pipeline

Figure 5.4 illustrates the processing pipeline for effecting high-level optimizations. There are two new tasks for the compiler to perform: the generation of rewrite rules from axioms and the application of the generated rewrite rules to transform code. To accomplish the first task, the axioms in concepts are extracted during parsing and type-checking, then instantiated for specific types along template instantiation, and finally interpreted as rewrite rules to be stored into a "rule repository." The rules in this repository are available for the ME's rewrite engine. The second task is the responsibility of the *function abstraction analyzer*, which governs function inlining and rule application, as well as determines which rules to attempt in each function so that excessive attempts of rule application are avoided.

Generating Rewrite Rules

Generating conditional rewrite rules from axioms is achieved by leveraging the processing of concepts that is already part of ConceptGCC. Structurally, a concept is very similar to a C++ class template. Indeed, ConceptGCC internally represents concepts as class templates (axioms as member functions), and concept maps as specializations of those class templates, [12, 14]. Instantiating an axiom for particular concrete types, as a result of processing a concept map definition, is thus accomplished by way of normal template instantiation. For example, the Identity axiom in Figure 5.1, when instantiated as part of a concept map for Monoid<plus<string>, string>, yields the *axiom instance* in Figure 5.5.

One rule in an axiom instance is a single expression. The pattern and the condition(s) in a conditional rewrite rule are, however, composed of a group of expressions that are chained together by data-flow information. For example, the rule $\mathcal{R}6$ contains two expressions, x + y and string(""), where y in the first expression is defined by

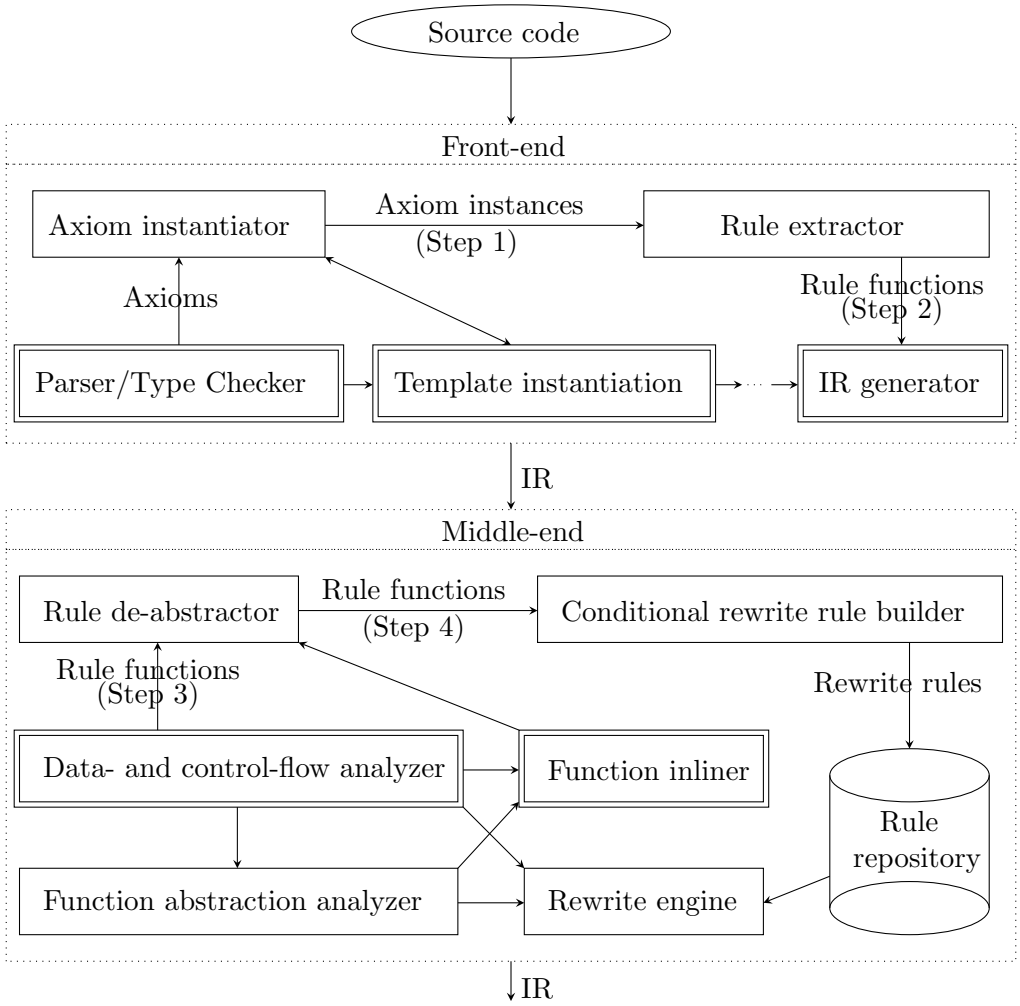

Figure 5.4. The processing pipeline for effecting high-level optimizations. The boxes with single lines represent functional units related to processing of high-level rewrite rules. The boxes with double-lines represent functions that are part of a typical compiler—we only show the functions relevant to our framework. The arrows indicate data dependencies (flow of data) between functional units.

```
axiom Identity (plus<string> op, string x) {
  op (x, Monoid<plus<string>, string>::identity_element (op, x)) == x;
  op (Monoid<plus<string>, string>::identity_element (op, x), x) == x;
}
```

Figure 5.5. Axiom instance generated from the Identity axiom in Figure 5.1. The qualifier Monoid<plus<string>, string>:: preceding the identity_element function identifies the concept map for which the axiom is instantiated.

the second expression. In order to bridge the structural difference between a rule in an axiom instance and its corresponding conditional rewrite rule, the sub-expressions in this rule is recursively factored out by substituting each sub-expression with a temporary variable. The result is a representation of the LHS and RHS expressions of the rule in a three address form. In GCC this form is called GIMPLE [92]. Afterwards, the conditional rewrite rule for the original rule in the axiom instance is obtained by performing data-flow analysis and necessary transformations as described below. In detail, this process of generating conditional rewrite rules involves four steps of transformations to axiom instances.

Step 1 extracts rewrite rules from an axiom instance. Since a function is the basic processing unit in a compiler from parsing to code generation, each rule in the axiom instance is represented as a pair of functions, extracted and derived from each side of the rule by the *rule extractor* unit. As a naming convention, this discussion prefixes the names of such *rule functions* with "rule". So, the first rule in the axiom instance in Figure 5.5 gives rise to the pair of rule functions in Figure 5.6.

Step 2 translates a rule function into its intermediate representation and performs the subsequent control- and data-flow analyses for it. This step is part of the normal processing for a function in the compiler. For example, subject to GCC's standard translation from AST into the GIMPLE form [92], the rule functions in

```
string rule_string_identity_lhs (plus<string> op, string x) {
  return op (x, Monoid<plus<string>, string>::identity_element(op, x));
}
string rule_string_identity_rhs (plus<string> op, string x) { return x; }
```

Figure 5.6. A pair of rule functions representing the right identity rule in Figure 5.5.

Figure 5.6 turn into the from in Figure 5.7. On this form, the compiler performs control- and data-flow analyses for rule functions to reason about data flow facts.

```
string rule_string_identity_lhs (plus<string> op, string x) {
  t1 = Monoid<plus<string>, string>::identity_element(op, x);
  t2 = op(x, t1);
  return t2;
}
string rule_string_identity_rhs (plus<string> op, string x) { return x; }
```

Figure 5.7. Rule function's IR which results from processing the rule functions in Figure 5.6 in the normal flow of processing a normal function in GCC.

Step 3 eliminates the extra function abstractions that are present in an axiom instance but should not be used in rewrite rules generated from the axiom instance. For example, the Identity axiom in the Monoid concept is written in terms of the identity_element function. Each Monoid concept map specializes this function to some expression that will construct an identity element for a particular model of a Monoid; it is this specialized expression that is used in non-template user code, not the identity_element function. In the case of Monoid<plus<string>, string>, the identity_element function is specialized to the expression string("").

Another source of extra abstractions are function objects that wrap function symbols, such as plus<string>() in the model Monoid<plus<string>, string> which wraps the call to the operator+ function overloaded for string.

The *rule de-abstractor* unit identifies local functions in concept maps and/or the function calls resulting from the use of function objects, and applies function inlining to eliminate these abstractions. Thus, the rule functions in Figure 5.7 become those in Figure 5.8.

```
string rule_string_identity_lhs (plus<string> op, string x) {
  t1 = string("");
  t2 = operator+(x, t1);
  return t2;
}
string rule_string_identity_rhs (plus<string> op, string x) { return x; }
```

Figure 5.8. Results from eliminating a certain class of abstractions in the rule functions in Figure 5.8.

Step 4 constructs the rule's LHS and RHS patterns from their corresponding rule functions and puts them into the rule repository. The pattern of each side of a rule is a AST-like tree, where the intermediate nodes represent function calls, or unary or binary operations, and the leaf nodes constants or *free variables*, i.e., function parameters. Constructing such a pattern is obtained by examining the control-flow graph (CFG) of a rule function, with the assumption that data-flow analysis has computed the use-definition information for the CFG. Starting from the last expression in the CFG (i.e., the argument of the return statement in the rule function), we recursively process each expression as follows. If an expression represents a function call (or an operator invocation), we create a pattern node for this call and link the arguments' patterns (constructed recursively) to the node; if an expression is a temporary (all temporaries are generated by the compiler), we replace

this temporary with the pattern constructed by processing its definition in the CFG; if an expression is a constant, the constant is the pattern. As an example, consider the rule function rule_string_identity_lhs. Figure 5.9(a) shows its CFG. Applying the pattern construction strategy on its CFG produces the tree in Figure 5.9(b), where x is a free variable and "" a constant. This tree corresponds to the LHS of the rule $\mathcal{R}6$.

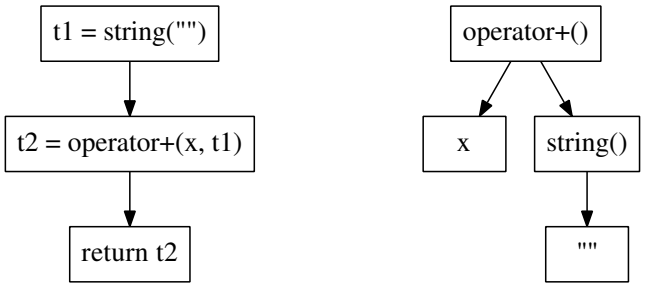

Figure 5.9. The CFG for the rule_string_identity_lhs rule function (a), and the pattern derived from it (b).

Applying Rewrite Rules

Applying a rule to a function is accomplished by downward traversing the function's CFG and employing the following strategy to each statement of the function. Given a statement, the rule's pattern is matched against the statement's AST. As the pattern and the statement both represent expressions, this matching process is essentially attempting to finding a substitution which maps the free variable(s) in the pattern to the expression(s) in the statement, known as Robinson's unification algorithm [100]. Note the case of attempting to match against a variable in the statement—if this variable itself does not lead to a successful match, we replace it with the expression defining it and continue the matching process (if this variable

has more than one definition, we simply give up the matching). A successful match is followed by replacing the redex denoted at the returned match position in the statement's AST with the appropriately substituted RHS of the rule.

The above strategy is sufficient, e.g., to uncover the rewriting opportunities in the code in Figure 5.3(a) and 5.3(b). To reveal the rewriting opportunity in the code in Figure 5.3(c), the rewriting approach is combined with function inlining.

Transformations and Inlining

Inlining plays two contradictory roles in application of rewrite rules. One the one hand, inlining can expose new facts justifying rewriting. E.g., inlining add() in Figure 5.3(c) reveals the fact that def(b) = string(""), which justifies the application of the rule $\mathcal{R}6$. On the other hand, too early inlining may lead to the loss of rewriting opportunities. For example, in Figure 5.3(c), if the constructor string() or the function operator+() is first inlined, the rule $\mathcal{R}6$ no longer matches. Thus, a strategy for interleaving inlining and rule application is necessary.

Compilers tend to carry out inlining in one pass, choosing candidates (functions to be inlined) and inlining them all at once. It would be prohibitively expensive to try a large number of different inlining orders, and with each inlining step, attempt to apply rewrite rules. To help reduce the search space of potentially useful inlining orders, *function abstraction analysis* is designed to obtain a measure, the *abstraction index*, of how "abstract" each function is in relation to other functions.

The abstraction index of a function is obtained from the program's call graph. Built-in functions and operations are at the lowest abstraction level, and a (non-recursive) function is always on a higher abstraction level than any of its callees. Concretely, the abstraction index ϕ of a built-in function is 0. For a non-recursive, user-defined function f, it is defined as:

$$\phi(f) = \max(\phi(g_1), \ldots, \phi(g_n)) + 1,$$

where g_i are the callees of f (and no g_i is f).

Computing the value of ϕ for a given function follows the depth-first traversal order of the program's call graph. Recursive functions are handled by keeping track of what functions have been visited and then ignoring recursive calls. More precisely, if a call path leads to a cycle from a caller to itself, then none of the nodes in that call path contribute to the computation of the abstraction index. The net effect is that recursive calls to a function have no effect on computing the function's abstraction index. For example, in the code in Figure 5.3(c), assuming the abstraction indices of the constructor string() and function add() are, respectively, 1 and 2, then $\phi(\text{main}) = \max(\phi(\text{string}()), \phi(\text{add})) + 1 = 3$.

The functions in a program, in particular those identified as candidates for inlining, can be partitioned based on their abstraction indices. Starting from functions in the partition with the highest abstraction index, each candidate is inlined and attempted to match the rewrite rules in the body of the just inlined function. The process is then repeated recursively for the functions in the partition with the next lower abstraction index. With this strategy, the rewriting effort becomes proportional to the number of partitions.

Using the abstraction indices of functions to guard rewriting allows further improvement on the efficiency of applying rewrites. Naively, each rewrite rule could be attempted to each function. In practice this is not necessary. It is possible to rule out many rewrite rules based on their rule functions' abstraction indices.

Given a function f and a rule function r, if $\phi(f) < \phi(r)$, the rule corresponding to the rule function r cannot match an expression in f. This property suggests a practical approach to combining inlining and rewriting.

The approach consists of one preprocessing and two rewriting phases. In the preprocessing phase, this approach partitions all functions that are candidates for inlining according to their abstraction index, and does the same to all rewrite rules. This approach orders the partitions of the function candidates and the rewrite rules,

respectively, in the descending order of their abstraction indices. In the first rewriting phase, then, this approach attempts to apply each rewrite rule to each function whose abstraction index is at least that of the rule function. In the second rewriting phase, it iterates over the ordered partition of the rewrite rules, interleaving inlining and rewriting operations. Specifically, in each iteration, it uses the rewrite rules from the partition as potential rewrite rules, inlines those function candidates whose abstraction indices are greater than or equal to the abstraction index of the potential rewrite rules, and attempts to match the potential rewrite rules to the bodies of the just inlined functions.

The above approach is practical, as demonstrated by the experiments described in Section 5.2. The number of iterations in the second phase is limited by the highest abstraction index of any of the rule functions, and each iteration has a set of rules to apply that is disjoint from the sets of other iterations.

5.2 Evaluation

This section describes the evaulation of the optimizing effectiveness of the approach of exploiting concepts and axioms for domain-specific optimizations in the middle-end of the compiler and the impact this approach has on the compiling effort. The prototype of the approach, implemented as an extension of the ConceptGCC compiler, can be obtained from our project home pages [101]. The prototype adds a command line option "`-fconcept-simplify`" for users to switch on the concept-based optimizations. In the following test runs, the exception mechanism was disabled with "`-fno-exceptions`" and the optimization switch was "`-O2`". The evaluation platform was an iMac 2GHz Intel Core Duo, running Mac OS X 10.5.3.

To measure the effectiveness of the approach, we selected programs that contain algebraic expressions that an optimizing compiler routinely simplifies if the arguments of those expressions are of built-in types. In these programs, built-in types were repalced with user-defined types whose operations obey the same algebraic laws

that justify the simplifications on built-in types. The evaluation aims to measure the *abstraction penalties* of these programs. Abstraction penalty is defined as the ratio of the execution time of an abstracted implementation over a direct implementation [1, §D.3]. The test programs are from Adobe's C++ performance benchmark suite[1] [16], designed to measure, among other traits, abstraction penalties of C++ compilers.

The benchmark wraps a varying number of double values into user-defined classes that support arithmetic operations (and thus follow the same algebraic rules as double) and executes code that repeatedly evaluates arithmetic expressions on objects of those classes. Figure 5.10 summarizes the results. The user-defined classes, DoubleClass, Double2Class, and Double4Class wrap one, two, and four doubles, respectively. Each test was repeated for each of these classes. The names of the tests indicate the algebraic operations being tested. As an example, the "mixed algebra" tests measure the optimizer's efficiency for compound expressions that include more than one kind of algebraic operations. The code for these tests, where T is a placeholder for one of the above three user-defined classes, is as follows:

```
T test (T input) {
  return (−(T(0) − (((input + T(0)) − T(0)) / T(1)))) ∗ T(1);
}
```

The baseline for the test of abstraction penalty measurement in this case is the function T base (T input) { return input; }.

The abstraction penalty is consistently essentially one with our optimizer. When the concept-based optimizations were switched off, on the other hand, the compiler only got rid of the abstraction overhead in three of the twelve cases. Our optimizations do not significantly slow down compilation. For this benchmark, applying the optimizations increases the compilation time by a factor of 1.0035.

[1]The benchmark suite's current public release does not yet include the tests used in this evaluation; the tests are available on the project home pages [101].

Test	D1(A)	D1(B)	D2(A)	D2(B)	D4(A)	D4(B)
add zero	1.00	1.25	0.99	1.65	1.01	1.70
zero minus	1.00	1.32	0.99	1.89	1.01	1.79
times one	1.00	1.00	0.99	1.00	1.00	0.99
mixed algebra	1.03	1.63	0.99	2.36	1.00	2.42

Figure 5.10. The benchmark results for algebraic simplifications for user-defined types. D1, D2, and D4 denote DoubleClass, Double2Class, and Double4Class, respectively. The columns denoted with (A) show the abstraction penalties measured with high-level optimizations on, the columns (B) show the same measured with those optimizations off.

What is the burden for the programmer to enable the identity rules for the user-defined types used in the benchmark? Recall that the identity rules are defined in the concept Monoid, which is predefined in a header file. The necessary concept map for the additive monoid for DoubleClass below is an example of one of the nine concept maps (an additive, subtractive, and multiplicative monoid for each of DoubleClass, Double2Class, and Double4Class) that the programmer would write to enable the identity laws:

```
concept_map Monoid<plus<DoubleClass>, DoubleClass> {
  DoubleClass identity_element (plus<DoubleClass> op, DoubleClass x)
    { return DoubleClass(0); }
};
```

To measure the impact of high-level simplifications to later analysis and optimization passes of the compiler, we estimated the size of the compiler's intermediate data at various stages of the compilation by measuring the size of the output GCC generates for each compilation stage when invoked with the option `-fdump-tree-all`. We used the above benchmark as our test program. Results of this measurement are shown in Figure 5.11. In the beginning, annotations for high-level optimizations, concepts and concept maps, increase the size of the representation, but during fur-

ther stages, the size decreases: high-level rewrites, applied early, reduce the workload of the later phases of the compilation.

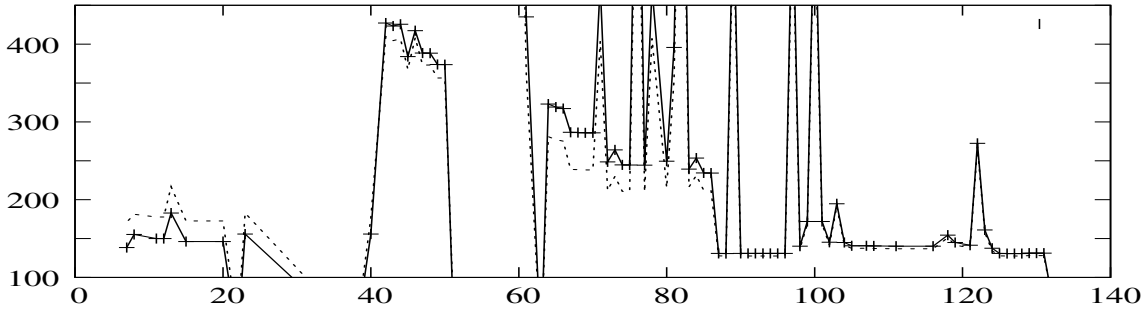

Figure 5.11. The impact of early high-level transformations on the size of intermediate data throughout compiling our benchmark. The horizontal axis enumerates the compilation passes in chronological order, the vertical axis denotes intermediate data size in kilobytes. The dashed line was obtained with `-fconcept-simplify`, solid line without it.

The performance of the motivating introductory example shown in Figure 5.3(c) was also measured. This example code requires appropriate inlining strategy to uncover the optimization opportunity. To measure the execution time, a loop invoking the add(x, y) function was iterated 100,000 times. The running time with the high-level optimizations turned on was 0.013 seconds, compared to the 0.015 seconds when they were turned off. To exercise the left identity rule, the experiment was repeated for the call add(y, x). Now the running times were 0.013 with high-level optimizations, and 0.033 without.

5.3 Conclusion

This chapter applies generic programming and the "concepts" language feature of C++ for realizing generic user-defined simplifications. A programmer specifies generic rewrite rules with axioms in concepts, and the rules are put to use for a particular type

by simply stating that this type models a particular concept. Such simplifications are effective for two reasons. First user-specified rewrite rules are transformed into conditional rewrite rules where data-flow facts (in the user-defined rewrite rules) are decoupled from rewrite patterns as the associated conditions with these patterns. Rule application becomes pattern matching against rewrite patterns and satisfying the conditions with these patterns. Second these simplifications are combined with function inlining in an effective way. An appropriate order of function inlining helps uncover more optimization opportunities.

This chapter also shows that the ability to perform high-level user-defined optimizations can be built into an industrial strength compiler without distracting the compiler architecture in major ways, and that the increase in the compiling resources to support these optimizations stays small. Further, the experiments in this chapter demonstrate that generic rewrite rules which apply to a large class of user-defined types effectively eliminate abstraction penalties where standard low-level optimization techniques fail to do the same.

6. SUMMARY-BASED DATA-FLOW ANALYSIS THAT UNDERSTANDS REGULAR COMPOSITE OBJECTS AND ITERATORS*

Recall the motivating example that is elaborated via Figure 1.1 and Figure 1.2, in the thesis's introduction. Figure 1.2 is reiterated below.

```
T x, y, z, w, r, s, t1, t2, t3, t4;
... // initializations
t1 = x + y
t2 = t1 + z;
r = t2;
... // code that does not change x and y
t3 = x + y
t4 = t3 + w;
s = t4;
```

If T in the above code is some built-in type, like int, a modern compiler will be able to perform a series of reasoning and transformation steps to the code: first x + y is identified as a common subexpression, then the use of t3 is replaced with the use of t1, and finally t3 = x + y becomes unreachable and is eliminated. The same reasoning, however, will not be possible if T is a user-defined type and even if it behaves as a built-int type. This chapter shows how with better understanding of the semantics of user-defined types, a compiler can perform more precise and scalable program analyses, thus enabling the discovery of more optimization opportunities for user-defined types.

This chapter focuses on *summary-based* analyses [62–64]. A procedure summary conservatively approximates a procedure, describing information such as the proce-

*Reprinted with permission from "Exploiting regularity of user-defined types to improve precision of program analyses", by Xiaolong Tang and Jaakko Järvi. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1743–1750, Trento, Italy, 2012. ©2012 ACM, Inc.
*Reprinted with permission from "Summary-based data-flow analysis that understands regular composite objects and iterators", by Xiaolong Tang and Jaakko Järvi, 2012. *ACM SIGAPP Applied Computing Review*, volume 12, pages 36–47.

dure's side effects and its impact to the points-to relation. Procedure summaries act as transfer functions at call sites. An analysis approximates the effect of a function call by binding a calling context to the function's parameters in the callee procedure's summary.

Generating a precise procedure summary is usually not possible; the knowledge about a procedure's *invisible variables*, i.e., objects accessible via the procedure's parameters and via global variables used in a procedure [65,66], is often incomplete. Wilson et al. argue [67] that computing a procedure's summary based on all possible aliases of its invisible variables is prohibitively expensive. Chatterjee et al. [62] use, in the context of object-oriented programming, the types of invisible variables to reduce the number of spurious aliases among the invisible variables: only if the type of one invisible variable is in a subtyping relation with the type of another invisible variable, the two variables may alias.

In many cases programmers know that many of the invisible variables conform to certain aliasing invariants and never alias with each other. This information often comes from the semantics of user-defined types. For example, two distinct objects of type std::vector<int> are not aliased; modifying one does not change the other. Traditional program analyses, however, do not make such aliasing assumptions [62].

Conveying the precise semantics of each different user-defined type to an optimizer is not feasible. This chapter employs a more practical strategy. This strategy identifies a set of common properties that many, possibly most, user-defined types possess, and that significantly improve the precision of aliasing guarantees in procedure summaries. In particular, the properties that are of interest are whether a type is *regular* or not, whether values of a type are *composite objects* or not, and the aliasing guarantees that follow from these properties. Both of these notions originate from generic programming [102, §12], and are in use in the C++ Standard Template Library (STL) [3].

Regular is one of the fundamental concepts appearing in the STL; types modeling this concept are *regular types*. They support default and copy construction, destruction, assignment, and equality comparison; total ordering is sometimes assumed, but for our purposes this requirement is not necessary. The operations of regular types conform to consistent semantics, described in Section 3.4. Examples of regular types include built-in types, such as int, char, and double.

When regularity is extended to aggregate types, the result is many user-defined types that have two useful aliasing guarantees: (1) an object "owns" the memory locations reachable through it and (2) two distinct objects, where one does not own the other, are not aliased. These guarantees are captured by the *composite object* concept[1] described by Stepanov et al. [102, §12.1]. This concept captures the "value semantics" that is common for most built-in types.

A composite object is composed of other objects, its *parts*. The objects accessible at constant offsets from the starting address of a composite object are its *local parts*; the other accessible objects from this address are its *remote parts*. A composite object owns its parts; if one composite object is not nested into another or vice versa, they are disjoint.

Non-aggregate built-in types like int, char, and double, are trivially models of the composite object concept[2]. Other examples include aggregate types that rely on the default construction, destruction, copying, and assignment semantics, and whose members are composite objects; arrays of composite object types; and STL container templates instantiated with composite object types.

This reasoning extends aliasing guarantees similar to those of built-in types to many user-defined types. For example, it is safe to assume that two distinct objects of std::vector<int> are disjoint. This chapter describes an abstraction for composite objects, and shows how the effort of points-to-analysis is reduced when types are

[1]Stepanov et al. call it a *concept schema* as it only pertains to non-syntactic properties of objects.
[2]C++ has features that allow casting a value of one type to another type, and thus circumventing the typing discipline. We assume that such features are not used here.

classified to be models of the composite object concept, and how the precision of procedure summaries is improved.

As containers are a common category of regular composite types, making the points-to analysis aware of the concept of *iterators* allows further reducing the pointer analysis effort. Iterators are abstractions of regular pointers and prevalent in modern C++; every standard library container, for example, provides an iterator interface for accessing the container's elements in some predefined order. Iterators complicate taking advantage of objects' compositeness, as they provide a direct access to objects' parts. These challenges, and solutions to them, are discussed in Section 6.3.

While the notions of regularity and composite objects originate from generic programming, the means to put them in use in compilers we borrow from object-oriented programming. Several works have focused on controlling aliasing between objects in object-oriented programs through type systems that maintain invariants about aliasing [71, 72, 103]. These works convincingly argue that many object types conform to aliasing invariants that could be exploited by compilers for analyses and optimizations. Most of these works, however, have not found their way to practice; production compilers do not recognize these invariants.

The contributions of this chapter are as follows:

- it characterizes the properties of composite objects and devises an economic abstraction for them;
- it designs a summary-based analysis based on the abstraction, in which an important part is modeling and exploiting the semantics of container-iterator relationships;
- it applies the analysis to uses of STL containers, and compared to traditional analyses observes more precise and concise procedure summaries; and
- it applies the analysis to three real-world applications, and observes that points-to relations and procedure summaries remain small.

The structure of this chapter is as follows. After the introduction, Section 6.1 explains summary-based pointer analysis. Section 6.2 details how regularity is exploited for program analyses. Section 6.3 describes how to leverage the well-established relation between iterators and containers for further improving program analyses. Section 6.4 presents a prototype that implements the ideas of this chapter and reports experiments. Finally, Section 6.5 concludes the chapter.

6.1 Summary-Based Analysis

Consider the code in Figure 6.1, part of a C++ implementation of a string class. This section uses the operator= procedure as an example to illustrate a typical summary-based analysis.

For precision, the analysis this chapter presents is field-, flow-, and context-sensitive. Field-sensitive analyses let one model each instance of a field as a separate object. Flow-sensitive analyses take into account the order of statements; an assignment to a pointer may kill the points-to relations that hold for the pointer before executing the assignment. Context-sensitivity enables distinguishing between the effect of different calls to a procedure, and as a result, heap objects allocated by the procedure can be kept distinct. Specifically, for context-sensitivity, each heap object is associated with a *call string* [53] as its identification. E.g., the heap object arising from the call at Line 20 is denoted as $\mathsf{heap}_{31\to 20}$.

This chapter also follows the convention of program analysis to abstract run-time objects of a program. An array is abstracted as a single object, the objects that are recursively accessible from an object are approximated with this object itself, and each dynamically allocated object is denoted by its allocation site. For instance, the allocated array at Line 31 and Line 33 are denoted, respectively, by heap_{31} and heap_{33}, and the array str[] is represented as just str.

```
class string {
  public:
  string(const string &s) {
    int l = s.rep→ len; rep = new (l) Rep(l); strcpy(rep→ str, s.rep→ str);
  }
  string(int l = 0) { rep = new (l) Rep(l); }
  string(const char *s) {
    int l = strlen(s); rep = new (l) Rep(l); strcpy(rep→ str, s);
  }
  ~string() { delete rep; }
  string &operator=(const string& s) {
    if (this == &s) return *this;
    int l = s.rep→ len;
    if (l <= rep→ max) {
      delete rep; rep = new (l) Rep(l);
    }
    strcpy(rep→ str, s.rep→ str);
    return *this;
  }
  void swap(string &s)
  { Rep *t = rep; rep = s.rep; s.rep = t; }
  char *find(char);
  private:
  struct Rep {
    Rep(int l) { len = l; max = l; str[0] = '\0'; }
    void *operator new(size_t s, unsigned long l)
    { return new char[s + l]; }
    void *operator new(size_t s)
    { return new char[s]; }
    int len; int max; char str[1];
  };
  Rep *rep;
};
```

Figure 6.1. Part of the definition of a string class

Summary-based analysis commonly consists of two phases. The first and main phase is computing the side effects and the points-to relations that a procedure may produce at any calling context, and representing these behaviors as the summary of the procedure. This phase is run as a bottom-up traversal of the call graph of a program. The second phase runs as a top-down traversal and propagates the actual arguments at a call site to its corresponding callee, and then computes the final points-to relations and side effects at each program point.

Given a procedure p, its summary is represented as follows:

$$\begin{aligned} Sum(p) &::= (Sum_{\text{pt}}(p), Sum_{\text{se}}(p)) \\ Sum_{\text{pt}}(p) &::= \{\overline{(C, (\eta, \alpha))}\} \\ Sum_{\text{se}}(p) &::= \{\overline{(C, (\alpha, e))}\}, e \in \{\text{RD}, \text{MOD}\} \end{aligned}$$

where the metavariable C ranges over sets of conditions (to be explained later), α over the objects accessible in p, and η over the pointers accessible in p; $\overline{(C, (\eta, \alpha))}$ means a comma-separated sequence of $(C_1, (\eta_1, \alpha_1)), \ldots, (C_n, (\eta_n, \alpha_n))$, and similarly for $\overline{(C, (\alpha, e))}$; RD and MOD denote the read and modification effects, respectively. $Sum(p)$ is represented as a 2-tuple. Its first part $Sum_{\text{pt}}(p)$ represents the set of points-to relations that p may produce, where an element $(C, (\eta, \alpha))$ means that under C the pointer η refers to the object α. Its second part $Sum_{\text{se}}(p)$ represents the set of side effects that p may produce, where $(C, (\alpha, e))$ means that under C the object α has the side effect e. Note that α and η are represented as access paths. An access path is of the form $\mathsf{v}\{.\mathsf{f}|.*\mathsf{f}\}$, where v represents an object, f a field in that object, $*\mathsf{f}$ the object obtained by dereferencing the field f, $.\mathsf{f}|.*\mathsf{f}$ either $.\mathsf{f}$ or $.*\mathsf{f}$, and $\{.\mathsf{f}|.*\mathsf{f}\}$ a sequence of zero or more instances of $.\mathsf{f}|.*\mathsf{f}$.

To analyze the operator= procedure, the invisible variables of the procedure are needed to be presumed first. Through the parameter this, String object may be accessed, and further a Rep object; and similarly for the parameter s. Figure 6.2

depicts the invisible objects we presume from the parameters. The presumption, however, is not precise. In Figure 6.2, string1 may be an alias to string2, because they have the same type. Similarly, rep1 may be an alias to rep2. An analysis has to take into account all these aliasing cases between the invisible variables when computing points-to relations at pointer assignments. Take as an example the assignment at Line 20 in Figure 6.1. If string1 is not an alias to string2, the assignment only causes the rep field of string1 to refer to the heap object $\mathsf{heap}_{31\to 20}$; otherwise, it also causes the rep field of string2 to refer to this heap object. To distinguish the different cases, each points-to relation is associated with a (possibly empty) set of conditions, where each condition describes an aliasing between two objects. Thus, the generated procedure summary for the operator= method, i.e., $Sum_{\mathrm{pt}}(\mathsf{operator}=)$, is:

$$\begin{aligned}
&\{(\emptyset, (\mathsf{ret}, \mathsf{string1}), (\emptyset, (\mathsf{string1.rep}, \mathsf{rep1})),\\
&(\emptyset, (\mathsf{string1.rep}, \mathsf{heap}_{31\to 20})), (\emptyset, (\mathsf{string2.rep}, \mathsf{rep2})),\\
&(\{\mathsf{string1} == \mathsf{string2}\}, (\mathsf{string2.rep}, \mathsf{heap}_{31\to 20}))\}
\end{aligned}$$

where ret denotes the return variable of the procedure, and string1 == string2 means that string1 is an alias to string2. Note that $Sum_{\mathrm{se}}(\mathsf{operator}=)$ is not shown here. Given the points-to relations at every program point of a procedure, it is straightforward to compute the side effects that the procedure may produce, as described by Landi et al. [54].

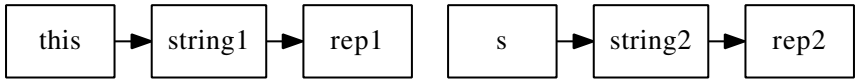

Figure 6.2. The invisible objects of operator=.

The problem with the above analysis is that it is not aware of the high-level properties known to programmers, and thus it may generate too conservative points-to relations. Consider analyzing a procedure with this signature:

```
void foo(std::vector<string> &v, std::deque<string> &d)
```

Figure 6.3 and Figure 6.10 depict the presumed invisible objects from the parameters v and d, respectively, when using the vector and deque implementations in the GCC's standard C++ library. As described above, making the rep field of string3 point to a new heap object gives rise to conditional points-to relations for the other invisible string objects. For example, the rep field of string8 may also refer to that same heap object if string3 == string8. The programmer knows, however, that string3 cannot be the same as string8. Any object of type std::vector<string> is disjoint from any object of type std::deque<string>, thanks to the semantics of these types, as discussed in Section 6.2.1. Therefore, string3 accessed from v and string8 accessed from d are disjoint.

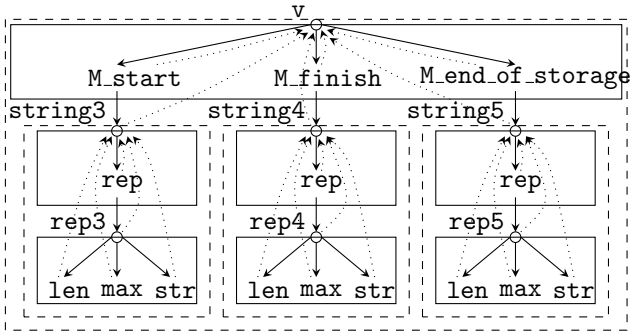

Figure 6.3. The invisible objects from the parameter v in the foo procedure. The solid boxes denote objects; the circles denote the starting addresses of objects; the dashed boxes denote the imaginary boundaries of the interiors of composite objects; the solid arrows denote the references to fields or objects; and the dotted arrows indicate the innermost owners of the parts of composite objects.

Further cases where the analysis may fail to preserve the knowledge of the disjointness of objects include the use of swapping, merging, or transferring operations between objects, and the use of the copy-on-write technique [104]. As an example of how the swap operation obscures the analysis, assume that x and y are of type string from Figure 6.1. Analyzing now, say, the statement if(...) x.swap(y); would not reveal that x and y are disjoint.

The empirical data which is shown in Table 6.1 confirms that the number of points-to relations at a program point (computed using the above analysis) tends to be large. Large points-to relations impair the scalability of program analysis—applying procedure summaries at call sites is expensive. Applying a procedure summary means computing all possible actual-formal parameter binding lists at the call site to the procedure; each binding list is a one-to-one mapping from each formal parameter and invisible object of the procedure to its corresponding value at the call site. A large points-to relation at a call site may result in a large number of possible actual-formal parameter binding lists.

Many points-to relations only describe the internal state of composite objects. When such points-to relations are abstracted away, the number of points-to relations kept at program points is significantly reduced, and thus also the number of possible actual-formal parameter binding lists.

As an example, consider the following code snippet:

```
40  string(''Hello world'') x;
41  string(''Bye bye'') y;
42  x = y;
```

Analyzing the code reveals that x.rep may point to two heap objects, $\mathsf{heap}_{31\to 11}$ and $\mathsf{heap}_{31\to 20\to 42}$. Keeping both points-to relations is expensive. Both of the above two heap objects, however, are encapsulated in x and are accessed via a common access path relative to x. Thus, the two heap objects can be removed, and references to

them replaced with a reference to x. All points-to relations with the removed heap objects as targets can be dropped.

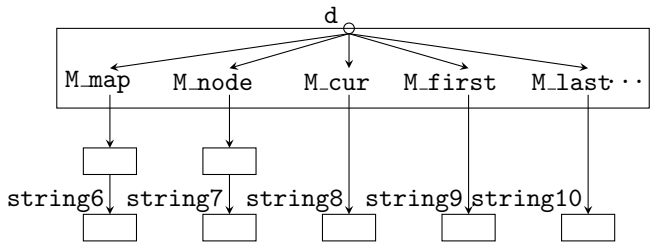

Figure 6.4. The invisible objects from the parameter d in the foo procedure. M_map and M_node are of type string$**$, and the other three fields depicted are of type string$*$. For simplicity, we omit another three fields of type string$*$ and one field of type string$**$ in d and hence the invisible objects from these fields.

6.2 Exploiting Regularity

Regular types provide a base for precisely understanding the behavior of user-defined types. The concepts feature in ConceptC++ provides a natural way to declare a type to be regular; in fact, Regular was a primitive concept in the concepts language extension. Without direct language support for concepts in C++11, one alternative is, for example, relying on C++11's attribute syntax [105]. For example, the declaration class string [[regular]] can assure the compiler that string class is regular.

6.2.1 Composite Objects

Regularity itself is too weak a property for the kind of analyses described in this chapter. When regularity is combined with guarantees about the relationships between an object and what it refers to, sufficiently powerful properties are obtained. To this effect, Stepanov et al. characterize the "composite object" concept, the mod-

els of which are called composite object types. Four properties relating to the object and its parts hold for all composite objects: *connectedness*, *noncircularity*, *disjointness*, and *ownership* [102, §12.1]. Connectedness means that parts are reachable from the object's starting address; noncircularity means that an object is not reachable from its starting address; disjointness means that an object exclusively owns its parts, allowing no partial sharing with any other object; ownership means that copying an object is *deep copying* (copying a pointer is accompanied with creating a fresh distinct copy of what the pointer refers to), and destroying an object destroys its parts too. Note that disjointness and ownership are conceptual requirements on composite objects; it is, for example, possible to design a composite object type that uses the copy-on-write technique.

The connectness and noncircularity properties of a composite object derive this corollary:

Corollary 1 *The accesses starting from (the starting address of) a composite object are bounded.*

What is reachable from a composite object is referred to as its *interior*, and any object accessible from it as its *component*. A component of a composite object can thus be the object itself, one of its local parts, or one of its remote parts.

The disjointness and ownership properties of a composite object result in access restrictions to the interior of the object. Access to the components of a composite object is required to conform to the *owner-as-dominator* property [72].

Property 1 (owner-as-dominator) *A composite object is the owner of its interior. This owner dominates the accesses to its components from the outside.*

To support the iterator idiom in the object-oriented programming, an iterator, as an exception, is permitted to directly access the interior of the composite object that the iterator is associated with. Section 6.3 describes what this means to our summary based points-to analysis.

Composite object types are closely related to ownership types [72]. In ownership types the owner-as-dominator requirement is statically checked. Our work, however, requires that programmers assume the responsibility for correctly specifying the set of composite object types.

The owner-as-dominator property enforces a nesting structure among the components of a composite object which are composite objects as well. In Figure 6.3, identified composite objects include v, string3, string4, and string5. Therefore, every component of v has an owner, and the interiors of the three string objects are nested inside the interior of v.

6.2.2 Composite Object Abstraction

At run time, a composite object's interior may acquire new components or lose old ones. The access to the interior of a composite object is, however, limited to the access paths originating from the object itself. (The accesses to a composite object's components via its iterators are resolved into an access through the object, by points-to analysis.). This allows the use of the access paths relative to its owner to symbolically represent its components. All possible access paths to the parts of a composite object determine its *static topology*. The static topology for a composite object is a graph. A node in the graph denotes the address of a component of the composite object, and an edge denotes the access path from one node to another. In Figure 6.3, the solid arrows depict the static topology of v, an object of type std::vector<string>. Though not shown, each node of the static topology is identified by an access path from v. For example, v.∗M_start corresponds to the object denoted by string3, v.∗M_finish to the object denoted by string4, v.∗M_start.∗rep to the object denoted by rep3, and v.∗M_finish.∗rep to the object denoted by rep4. Note that two access paths in a static topology, e.g., v.∗M_start and v.∗M_finish, may alias each other.

As is common, points-to relations are used to track what objects are the components of composite objects. These points-to relations are consulted to resolve

accesses to composite objects during the course of side-effect analysis. Maintaining these points-to relations can be expensive. Since they only describe the internal state of composite objects, it is, however, not necessary to expose them to the clients of composite objects. One alternative strategy is computing the possible side effects on composite objects arising from assignments to the pointer components of composite objects. An assignment to a pointer component of a composite object means a modification to what is reachable from the pointer component (including the pointer component). The potential aliasing between the access paths in a static topology, however, complicates keeping track of what is reachable from the pointer component. To address the complexity of tracking points-to relations, this chapter exploits the nesting structure of a composite object to design a strategy to normalize the access paths originating from a composite object, such that the normalized access paths do not have any aliases.

Assuming that x denotes a composite object, f denotes a data field of a component of x, $topo(x)$ denotes the static topology of x, α and β denote access paths from x, and $owner(\alpha)$ denotes the innermost owner of the component(s) denoted by α. The function used for normalizing an access path is called *norm*. The following describes the normalizing process for a given access path, α.

- If α is of the form x, $norm(\alpha) = x$;
- If α is of the form $\beta.f$, $norm(\alpha) = norm(\beta).f$;
- Otherwise, α is of the form $\beta.*f$. The processing then runs a depth-first traversal on the subtree rooted at $owner(\beta.f)$ in $topo(x)$ and return as n the first node whose type is the same as, or is in a subtyping relation with, the type of the object denoted by α. Finally, $norm(\alpha) = norm(n)$.

Take as an example the composite object v in Figure 6.3. The access path v.∗M_start is already normalized. The access path v.∗M_finish is normalized to v.∗M_start, because v is the owner of v.M_finish, and v.∗M_start denotes an object that has the

same type as the object denoted by v.∗M_finish. Also, $norm$(v.∗M_end_of_storage) = v.∗M_start. Thus, the objects denoted by rep3, rep4, and rep5 are all abstracted into one access path v.∗M_start.∗rep.

6.2.3 Concept-Aware Program Analysis

A novel program analysis is designed to exploit the properties of composite objects; it is called "concept-aware program analysis". Compared to the typical summary-based analysis as described in Section 6.1, the concept-aware program analysis handles two cases differently. First, it does not generate invisible objects if they are identified as components of composite objects. Second, it does not produce points-to relations for assignments to pointer components of composite objects; these assignments are instead considered as modifications on what is reachable from the targets of them. For instance, in Line 20 in Figure 6.1, the result of the analysis is that string1.∗rep is modified. Note that since string1.∗rep dominates the access to its components, a modification to string1.∗rep also implies modifications to its components.

The concept-aware analysis assumes that there is no aliasing between two distinct composite objects. Temporary aliasing, however, may occur as the result of transferring components between composite objects, or with the copy-on-write technique. Such temporary aliasing does not, however, lead to writing into two composite objects at the same time. Thus, the analysis respects flow- and anti-dependency between run-time objects of a program.

Correctness of Analysis

Conventionally, a safe abstraction strategy requires that one concrete object corresponds to at most one abstract object. If this is not the case, program analysis itself must maintain the consistency between all abstractions that correspond to the

same concrete object. In general, maintaining such consistency is challenging, because during program analysis it may not be known which abstract objects alias which other abstract objects. This issue has to be addressed, since the concept-aware program analysis may introduce two or more abstract representations for one concrete object. However, the scenarios in which this can happen are restricted, and can be handled as described below.

Specifically, two or more different abstract representations for one concrete object arise when a heap object or an invisible object is assigned to be part of a composite object. Figure 6.5 depicts a program path where an object is acquired in this way as

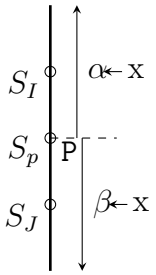

Figure 6.5. A program path where the concrete object x becomes part of a composite object at program point P. Before P, x is abstracted as α and after P, x is abstracted as β. S_P denotes the statement at P, S_I denotes any statement which may modify x before P, and S_J denotes any statement which may read x after P.

part of a composite object. In this figure, x corresponds to two abstract objects, α and β. β denotes an access path relative to the composite object which acquires α as its part, and in general α and β are two distinct representations. Figure 6.6 presents a concrete example. Figure 6.6(b) illustrates an abstract object heap1 which will be acquired as part of a composite object. After assigning p to rep, as in Figure 6.6(b), heap1 is no longer available, and all accesses via p become accesses to str1.$*$rep.

Even though the above scenario can lead to two or more abstractions for one object, their scopes during the analysis are always disjoint, and it is thus not necessary

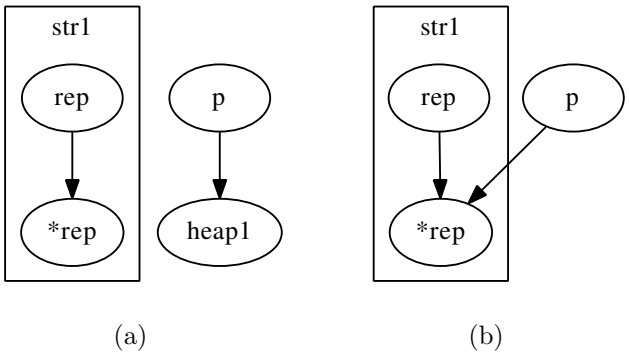

Figure 6.6. A heap object becomes part of a string, as a result of an pointer assignment str1.rep = p. Case (a) depicts the topology of the objects before the assignment and Case (b) after.

to explicitly maintain their consistency. To show the correctness of our approach, the following proves that, at any program point, the abstraction of all concrete objects contains the current value of the concrete object.

Consider again Figure 6.5. After the program point P, the concept-aware analysis makes α unavailable, and any modification to x after P is via β. Thus, if β is guaranteed to have the latest value of x at P, then β has the latest value of x at all program points after P. As described in the beginning of Section 6.2.3, the analysis handles a pointer assignment where the target is a component of a composite object specially, considering it as a modification to all objects accessible from the target. For example, in Figure 6.6, assigning p to rep results in modifications to all objects accessible from rep. This strategy forces a dependency between α and β. This dependency ensures that the modifications on α before P are also observed on β, and β thus has the latest value of x at P.

The above described strategy guarantees that the data dependencies of a program are preserved. Consider again the path in Figure 6.5. Since S_I may modify x before P and S_J reads x after P, then S_J depends on S_I, denoted as $S_J \lhd S_I$. Because the analysis forces $S_J \lhd S_p \lhd S_I$. Thus, $S_J \lhd S_I$.

The points-to and side-effect analysis result of the analysis is applicable to other analyses and many optimizations. Care must be exercised, however, with liveness analysis, specifically when computing the liveness information for an abstract object which is acquired as part of a composite object. Consider the abstract object α in Figure 6.5. This object α seems not to be alive after the program point P; this is not correct, however, because β aliases α. To safely compute the liveness information for an abstract object acquired as part of a composite object, the object's lifetime is conservatively extended to the end of the function where it is used.

Complexity of Analysis

The concept-aware analysis builds on the approach by Chatterjee et al. [62], and the worst case complexity results established for their analysis apply to ours as well. The work of this chapter exploits the semantics of user-defined types to improve the practical applicability of points-to analysis. The experiment in Section 6.4 demonstrates that the concept-aware analysis can be applied to large, practical applications.

6.3 Iterator Idiom Support

Requiring a composite object to strictly conform to the owner-as-dominator property is too restricted. To allow iterators to directly access the inside of composite objects, the owner-as-dominator property is compromised. This compromise presents a challenge for pointer analysis.

Without loss of generality, consider the case where a pointer may directly refer to the inside of a composite object. Figure 6.7 shows a code snippet where a pointer is allowed to directly access the character elements of a string. At Line 52, the return of the find call is assigned to q; thus q may refer to the inside of str1, as shown in Figure 6.8(a). An issue arises when str1 and str2 are swapped and q is continuously

```
50  string str1, str2;
    ...
52  char *q = str1.find('c');
    if (!q) {
54    str1.swap(str2);
      *q = 'd';
56  }
```

Figure 6.7. Code sample containing the use of a pointer to directly access the inside of a composite object. This code uses the string class defined in Figure 6.1.

used at Line 54 and Line 55[3]. Under normal pointer analysis, swapping str1 and str2 does not cause updating what q may refer to. Therefore, the behavior of the assignment at Line 55 is modifying str1.*rep.str rather than str2.*rep.str; this is not correct. To address the issue, it is necessary to update what q may refer to at Line 55, e.g., by making q conservatively refer to the inside of str1 and str2.

If a pointer may directly access the inside of a composite object, tracking what this pointer may refer to is not easy. Thus, using pointers to directly access to the inside of composite objects is prohibited. One exception is iterators. The user is required to provide knowledge on how to update what containers an iterator may refer to based on the behavior on the containers this iterator may be associated with. Consider the string class in Figure 6.1. Suppose that the iterator support for the string class is provided by defining a nested iterator class and necessary member methods for the string class. The specification for this iterator class may be as follows:

- string::iterator may access a string directly.
- If a string iterator may refer to a string prior to a statement and the rep field of this string may be modified by this statement, then, after this statement,

[3]The C++11 standard specifies the behavior of the swap member function of a container as follows: "Every iterator referring to an element in one container before the swap shall refer to the same element in the other container after the swap." [79, §23.2.1]

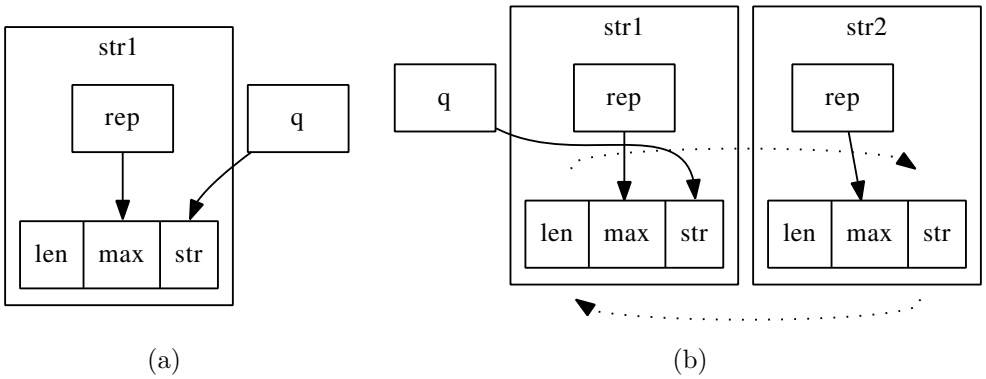

Figure 6.8. These figures are to demonstrate that we fail to update what an iterator may refer to in some cases if iterators are allowed. Case (a) shows that the assignment at Line 52 in Figure 6.7 gives rises to q $\rightarrow$ str1.$*$rep.str; Case (b) shows that q remains to refer to str1.$*$rep.str after the swapping operation at Line 54 in Figure 6.7, which is denoted by the dashed arrows. Note that q shall refer to str2.$*$rep.str in Case (b).

this string iterator may refer to all strings whose rep fields are modified by this statement.

The updating strategy in the specification may lead to conservative points-to relations for string iterators. This strategy is, however, feasible. First, iterators are usually local variables, so the complexity of managing iterators is local as well. Second, most operations on containers do not result in updating the iterators associated with them.

6.3.1 Exploiting Iterator-Container Relationship

Recognizing composite objects can significantly reduce the number of aliasing relations in procedure summaries; Section 6.4 reports observed reductions that result from leveraging knowledge about composite objects. For further reductions, the aliasing properties of iterators are studied. This is justified, because the C++ standard library (as well as many other C++ libraries) follow a well-established container/it-

erator abstraction that provides uniform semantics for different kinds of containers and iterators; analysis that relies on the generic properties of iterators thus applies widely.

Because iterators generally are not composite (an iterator does not own what is accessible from it), analyzing functions with iterator parameters is expensive. Consider analyzing the std::copy template, instantiated for std::deque<string>::iterator, shown in Figure 6.9. Figure 6.10 illustrates the invisible variables that may be accessed from the parameter first; similarly for the other parameters. The large number of the different possible aliasing relations amongst the invisible variables of type string makes the analysis for the std::copy function expensive.

```
typedef std::deque<std::string>::iterator iter;
void copy(iter ret, iter first, iter last, iter result);
```

Figure 6.9. The std::copy algorithm instantiated with deque<string>::iterator as the template argument; note that return value optimization [106] turns the return value into the first parameter of this instance.

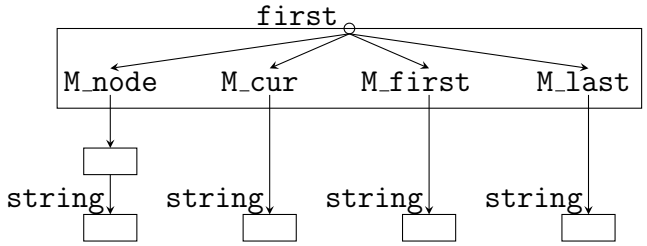

Figure 6.10. The invisible variables derived from a deque<string>::iterator parameter. M_node, M_cur, M_first, and M_last are the four members of a deque iterator. M_node is of type string** and the others of type string*. A string is composite, and thus we do not derive invisible variables from it.

Two observations about containers and iterators provide insight into the strategy which is used for improving the analysis for functions taking iterators as inputs. First, an iterator is associated with a container, and accesses via an iterator are essentially accesses to its associated container. Second, the points-to relation between an iterator and its associated container is established by the begin() method for the associated container. Thus, the program analyses translates accesses via an iterator to the accesses to its associated container, if the associated container is composite.

The strategy is as follows. The compiler is conveyed the information on which composite containers are associated with what iterators, and which of the containers' methods establish the points-to relations from iterators to the container. That information is generally available from the common structure of containers; given a container T, T::iterator and T::const_iterator are its iterators, and T::begin() decides how the two iterators refer to T. Thus, the compiler is made aware of the common knowledge between containers. This information is used in the following circumstance.

Given an iterator parameter, its associated container and the begin method of the associated container are decided with the help of the common knowledge between containers. If the associative container is composite, an invisible variable is derived to represent this container. Then the summary of the begin method is utilized to establish the points-to relations between the iterator parameter and the newly derived invisible variable.

Consider again analyzing the std::copy instance in Figure 6.9. The parameter type of std::copy indicates that it is associated with a composite container. Thus four invisible containers are generated, one for each of the four iterator parameters. Representing the containers explicitly, as invisible objects, significantly reduces the cost of analyzing this std::copy instance. This strategy only needs to derive four invisible variable. In contrast, without concept-aware analysis potentially 24 invisible variables are generated, 6 for each input. This strategy eliminates the need for creating any points-to relations; otherwise, a few hundreds of points-to relations have to be

maintained. As can be seen in Table 6.1, exploiting the iterator-container relationship enables a successful analysis of the operator= method of the std::deque<std::string>. This function's body contains the use of the std::copy function.

In addition to the begin() function, there are of course other functions that construct iterators. These can similarly be deduced in the high-level semantic knowledge of the concept-aware analysis.

6.4 Experiments

The prototype implementation of the concept-aware program analysis is built on the LLVM [17][4], and it conducts the first phase of a summary-based analysis as described in Section 6.1.

To evaluate the analysis, the STL is used as one benchmark. The evaluation computes the procedure summaries of the methods in five instantiated STL classes, using both the traditional analysis described in Section 6.1 and the concept-aware analysis. The comparison results in Table 6.1 indicate that concept-awareness significantly reduces the effort of analyzing the five instantiated classes.

Table 6.1 is divided into five groups, each for the methods of one class; the table does not include private methods or methods whose procedure summaries are trivial. In each group, each row compares the procedure summary of a method generated by the traditional analysis and the procedure summary of the same method generated by the concept-aware analysis. The notations in this table are as follows. Columns I and II denote, respectively, the results from the traditional analysis and the concept-aware analysis; the reduction, in percents, from I to II is shown in columns III. #args denotes the number of function arguments; PTA denotes the set of points-to relations in a procedure summary or at a program point; PTB denotes the subset

[4]The current implementation is built on Revision 126844 of LLVM SVN [17]. The code is accessible at `http://parasol.tamu.edu/groups/pttlgroup/high-level-optimization/index.html`.

Method	#args	#PTA			#PTB			#PTC			#SDA			#SDB		
		I	II	III	I	II	III	I	II	III	I	II	III	I	II	III
`std::string`																
ctor	2	3	0	100%	2	0	100%	1	0	100%	5	1	80%	9	3	67%
operator=	2	5	1	80%	2	0	100%	1	0	100%	6	1	83%	11	3	73%
insert	3	4	1	75%	2	0	100%	1	0	100%	6	1	83%	11	3	73%
reserve	2	2	0	100%	1	0	100%	0	0	n/a	5	1	80%	8	2	75%
begin	1	3	0	100%	2	0	100%	0	0	100%	6	1	83%	9	2	78 %
append	2	5	1	80%	2	0	100%	1	0	100%	5	1	80%	10	3	70%
swap	2	8	0	100%	4	0	100%	2	0	100%	7	2	71%	12	3	75%
`std::vector⟨std::string⟩`																
ctor	2	12	0	100%	7	0	100%	3	0	100%	8	3(1)	62%	14	6	57%
operator=	2	21	1	95%	11	0	100%	6	0	100%	11	5(1)	55%	18	9	50%
insert	4	22	0	100%	12	0	100%	9	0	100%	13	4(1)	69%	24	6	75%
swap	2	12	0	100%	0	0	n/a	6	0	100%	6	8(2)	-33%	6	8	-33%
resize	3	16	0	100%	7	0	100%	4	0	100%	11	3(1)	73%	20	6	71%
push_back	2	15	0	100%	7	0	100%	4	0	100%	11	3(1)	73%	20	6	71%
`std::deque⟨std::string⟩`																
ctor	2	24	0	100%	17	0	100%	6	0	100%	16	6(1)	63%	28	16	43%
operator=	2	*	2	n/a	*	1	n/a	*	0	n/a	*	6(1)	n/a	*	16	n/a
swap	2	34	0	100%	0	0	n/a	12	0	100%	8	12(2)	-50%	8	12	-50%
insert	4	114	10	91%	65	1	98%	64	0	100%	17	4(1)	76%	40	17	57%
push_back	2	19	1	95%	11	1	91%	1	0	100%	17	10(1)	41%	21	13	38%
erase	3	27	6	78%	4	0	100%	6	3	50%	19	13(5)	32%	30	18	40%
resize	3	65	1	98%	42	0	100%	24	0	100%	19	6(1)	68%	30	8	73%
`std::list⟨std::string⟩`																
ctor	2	6	0	100%	5	0	100%	0	0	n/a	7	1	86%	12	4	67%
operator=	2	5	1	80%	2	0	100%	0	0	n/a	8	3(1)	63%	13	6	54%
insert	4	2	0	100%	1	0	100%	0	0	n/a	7	2	71%	12	5	58%
splice	3	4	0	100%	0	0	n/a	0	0	n/a	4	5(3)	-25%	4	5	-25%
push_back	2	6	0	100%	5	0	100%	0	0	n/a	7	3(1)	57%	11	5	55%
swap	2	4	0	100%	0	0	n/a	0	0	n/a	4	4(2)	0%	4	4	0%
merge	2	6	0	100%	0	0	n/a	0	0	n/a	2	8(2)	-300%	6	10	-67%
resize	3	3	0	100%	1	0	100%	0	0	n/a	8	2(1)	75%	14	5	64%
`std::set⟨std::string⟩`																
ctor	2	13	0	100%	12	0	100%	0	0	n/a	10	3(1)	70%	18	9	50%
operator=	2	15	1	93%	12	0	100%	0	0	n/a	10	5(1)	50%	19	11	42%
insert	3	15	6	47%	11	6	45%	0	0	n/a	8	5(3)	38%	17	10	41%
swap	2	6	0	100%	0	0	n/a	0	0	n/a	8	10(2)	-25%	8	10	-25%
set_union	6	32	4	88%	28	4	86%	0	0	n/a	8	4(3)	50%	19	12	36%
set_intersection	6	14	4	71%	11	4	64%	0	0	n/a	7	4(3)	43%	17	12	29%
set_difference	6	23	4	83%	20	4	80%	0	0	n/a	7	4(3)	43%	17	12	29%

Table 6.1
Report on analyzing five instantiated STL classes.

of its corresponding PTA that have heap objects as sources or targets and PTC the subset of its corresponding PTA with non-empty conditions; SDA and SDB, respectively, denote the set of modified objects and the set of read objects in a procedure summary or at a program point. #PTA denotes the size of PTA, and the same notation applies to PTB, PTC, SDA, and SDB as well. * denotes unavailability of a measurement due to excessively long execution time of the analysis.

In concept-aware analysis, most procedure summaries have only one or two points-to relations; only a handful of points-to relations track aliasing inside composite objects (indicated by PTB in the table); and only one procedure summary contains conditional points-to relations (indicated by PTC). For those procedures whose inputs cannot be recognized as composite objects, e.g., the last three set operations in the table, our approach falls back to the traditional analysis.

The concise points-to relations in procedure summaries simplify the application of procedure summaries at call sites. In this benchmark, the traditional analysis fails (it gave up after half an hour on a modern desktop machine) on the operator= procedure of the deque<string> class, because applying the procedure summary of one of its callees is so expensive—there are over 10,000 possible actual-formal parameter binding lists for this application. Another benefit of the concise points-to relations is the reduced number of side effects in most procedure summaries.

To assess the precision of the side effects produced by our approach, the evaluation report unifies the components of each composite object involved in the write effects of procedure summaries. Table 6.1 also shows the number of the corresponding unified write effects (in parentheses). The concept-aware analysis reveals only one write effect on a composite object for most methods.

Further evaluation for the concept-aware analysis is conducted using several real-world applications: two programs (*252.eon*, *llvm-bcanalyzer*), and one library (*LiteSQL*). 252.eon is the only C++ benchmark in SPEC CPU2000. The benchmark

Program	max(#PTA)			max(#SDA)			max(#SDB)		
	I	II	III	I	II	III	I	II	III
252.eon	42	36	14%	78	76	3%	41	40	2%
llvm-bcanalyzer	249	99	60%	36	37	-3%	25	25	0%
LiteSQL	305	21	93%	45	26	42%	13	9	31%

Table 6.2
Report on points-to and side-effect analysis results at program points for three applications.

Program	max(#PTA)			max(#SDA)			max(#SDB)		
	I	II	III	I	II	III	I	II	III
252.eon	16	16	0%	78	77	1%	41	41	0%
llvm-bcanalyzer	233	49	79%	155	119	23%	103	104	-1%
LiteSQL	305	9	97%	21	26	-24%	58	20	66%

Table 6.3
Report on procedure summaries of the functions in three applications.

defines its own string class, and this class is the only composite object we recognize in the benchmark. llvm-bcanalyzer is a utility for analyzing bitcode files produced by LLVM-capable compilers. The easily identifiable composite objects involved in the utility are the STL string and an instance of the STL map. LiteSQL is a C++ library that provides object persistence into relational databases. The library is compiled into a single LLVM module, where the analysis recognizes the uses of the STL string, a few instances of vector, and one instance of both set and map. These all are composite objects.

Table 6.2 and Table 6.3 summarize the analysis results for the three applications, in two aspects; Table 6.2 is concerned with the points-to and side-effect analysis results at program points, and Table 6.3 the procedure summaries of functions. Table 6.2 and Table 6.3 use the same notations as in Table 6.1; besides, they introduce a new notation. max(#PTA) denotes the maximum size among a set of PTAs and such a notation applies to PTB, PTC, SDA, and SDB as well. For example, max(#PTA)

in Table 6.2 stands for the maximum among a set of numbers, where each number is the size of the set of points-to relations at a program point. For the moment, our prototype does not attempt to resolve function pointers and/or virtual function calls. Hence, both the traditional analysis and the concept-aware analysis fail to analyze some procedures. The traditional analysis also fails on these same procedures, and additionally on five more procedures, as there are too many possible actual-formal parameter binding lists (see below).

The concept-aware analysis does not notably improve the points-to analysis for 252.eon. This is because the string class defined in 252.eon is not involved as a building block for more complex data structures. The improvement on the points-to analysis for the other two applications, however, is significant. In particular, our approach thoroughly eliminates conditional points-to relations for LiteSQL and reduces the maximum size of its procedure summaries from 305 points-to relations to 9. The traditional analysis failed to analyze three procedures in llvm-bcanalyzer and two in LiteSQL; it gives up on a procedure if the number of possible actual-formal parameter binding lists for any call site in the procedure exceeded $10,000$. In the concept-aware analysis, the number of possible actual-formal parameter binding lists at all call sites is at most one; applying procedure summaries thus remains inexpensive.

6.5 Conclusion

This chapter shows how to take advantage of the properties of composite objects for computing more precise procedure summaries and side effects. The proposed concept-aware approach improves the scalability of a summary-based analysis. The experiments on STL classes indicate that the concept-aware analysis significantly reduces the number of points-to relations in procedure summaries. The side effects that the analysis discovers preserve the information that two distinct composite objects are disjoint. Analysis results of three relatively large applications suggest

that the analysis scales to practical use. The analysis thus provides a practical starting point for effective equational reasoning for user-defined types.

The code in Figure 7.1 contains optimization opportunities that exist because of the (algebraic) properties of user-defined types—and are typically beyond traditional compilers to exploit. The code defines a dynamic generic array, ggTrain, and a three-dimensional coordinate, ggPoint3. The initialize function takes as input an array representing a vector, and uses a loop to accumulate the componentwise absolute differences between the first element and every other element of the vector. To see

```
template <typename T>
struct ggTrain {
  T operator[](int i) const { return data[i]; }
  private: T *data; int nData, arraySize;
  ...
};

struct ggPoint3 {
  double& operator[](int i) { return e(i); }
  ggPoint3(const ggPoint3 &p)
  { e[0] = p.e[0]; e[1] = p.e[1]; e[2] = p.e[2]; }
  private: double e[3];
};

void initialize(const ggTrain<ggPoint3> &train) {
  double x = 0, y = 0, z = 0;
  int i = 0, j = 0;
  do {
    j++;
    x += fabs((train[j])[0] - (train[i])[0]);
    y += fabs((train[j])[1] - (train[i])[1]);
    z += fabs((train[j])[2] - (train[i])[2]);
  } while (j < train.length());
  ...
}
```

Figure 7.1. An example that illustrates optimization opportunities for user-defined operators. The code is extracted from the 252.eon program in SPEC2000, and slightly modified for better clarity.

the optimization opportunities in Figure 7.1, we examine the loop in the initialize function.

Generally a compiler simplifies complex expressions by converting them into three-address forms, e.g., via the "gimplifier" pass in GCC [107]. This takes place before attempting other transformations. For example, the simplified code for the loop in Figure 7.1 is shown (stylized) in Figure 7.2.

```
int i = 0, j = 0; int l1; double x = 0, y = 0, z = 0;
ggPoint3 t1, t2, t3, t4, t5, t6; double d1, d2, d3, d4, d5, d6;
double *p1, *p2, *p3, *p4, p5, *p6; double r1, r3, r5;
do {
  j++;
  t1 = train[j]; p1 = t1[0]; t2 = train[i]; p2 = t2[0];
  d1 = *p1; d2 = *p2; r1 = d1 - d2; x += fabs(r1);

  t3 = train[j]; p3 = t3[1]; t4 = train[i]; p4 = t4[1];
  d3 = *p3; d4 = *p4; r3 = d3 - d4; y += fabs(r3);

  t5 = train[j]; p5 = t5[2]; t6 = train[i]; p6 = t6[2];
  d5 = *p5; d6 = *p6; r5 = d5 - d6; z += fabs(r5);

  l1 = train.length();
} while(j < l1);
```

Figure 7.2. Code after breaking complex expressions in the loop in Figure 7.1. Several optimization opportunities can be observed in the simplified code.

There are three important aspects to notice in Figure 7.2. First, the numerous new variables, temporaries, introduced when breaking complex expressions. Second, the compiler has applied named return value optimization (RVO) [106] to the calls to operator[] in ggTrain. That is, t1 = train[j] is essentially a call to a function with the signature like this:

```
void operator[](ggPoint3&, ggTrain<ggPoint3>&, int);
```

This signature is not what results from instantiating the T ggTrain.operator[](int) template function. The compiler has added an additional parameter (the first) to rep-

resent the return value, and changed the return type to void. Third, no other transformations besides RVO have been applied.

The code in Figure 7.2 contains three kinds of optimization opportunities. First, train.length() is loop-invariant, because train is not modified in the loop. Second, since i is loop-invariant, t2 = train[i] and similar expressions are loop-invariant as well. Third, t2 = train[i], t4 = train[i], and t6 = train[i] are essentially common subexpressions, as are the similar expressions on train[j].

These three classes of optimization opportunities are not straightforward for a compiler to reveal.

First, proving that train.length() and t2 = train[i] are loop-invariant requires alias analysis, because train is essentially a pointer and accesses via a pointer may be aliased. Given sufficient aliasing information, train.length() may be proved to remain constant and not have side-effects, and thus may be hoisted out of the loop.

Second, the expression t2 = train[i] apparently modifies the object denoted by t2, and hoisting this expression out of the loop requires additional knowledge. Semantically the function call t2 = train[i] creates an object at t2, via the copy constructor of ggPoint3. This copy constructor must ensure that copy-constructing from the same value into t2 in the loop is invariant.

Third, in order to show that t2 = train[i], t4 = train[i], and t6 = train[i] are common subexpressions, a compiler needs to prove that t4, t6 are replaceable with t2.

Because of the lack of sophisticated pointer analysis, production compilers fail to exercise the first two kinds of optimizations. It is noted that even if a compiler applies such "de-abstraction" transformations as function inlining and replacing objects with their scalar members, the need for disambiguating the accesses via train (a reference) remains. For the running example, a typical compiler generates code that is equivalent to eliminating the common subexpressions in our concern, after the "de-abstraction" transformations. Consider t2 = train[i] and p2 = t2[0]. De-abstraction transformations for the two expressions generates code as below:

```
t2[0] = train.data[0][0];
t2[1] = train.data[0][1];
t3[3] = train.data[0][2];
p2 = &t2[0];
```

The variables in the above code are of built-in types; applying copy propagation, forward propagation, and dead code elimination eventually transforms the above code into a sole access to train.data[0][0]. Similarly, the expressions t4 = train[i] and p4 = t4[0] are transformed into an access to train.data[0][1]; the expressions t6 = train[i] and p6 = t6[0] are transformed into an access to train.data[0][2]. The behavior of these transformations is equivalent to removing t4 = train[i] and t6 = train[i] and replacing the the uses of t4 and t6 with t2.

Consider the aforementioned transformations which a typical compiler conducts to achieve the goal of eliminating the common subexpressions among the three expressions t2 = train[i], t4 = train[i], and t6 = train[i]. The particular definition for the copy constructor of ggPoint3 is crucial for the success of these transformations. Mutating the copy constructor, however, in some seemingly insignificant manner may challenge the compiler. E.g., consider another definition for the copy constructor:

```
ggPoint3(const ggPoint3 &p)
{ for (int i = 0; i < 3; i++) e[i] = p.e[i]; }
```

Unless loop unrolling, applied in a suitable order in relation to the other transformations, comes to rescue, a typical compiler fails to exercise the transformations discussed above. Other mutations to the definition of ggPoint3, e.g., using memcpy to copy data in its copy constructor or using dynamic memory for data, may ultimately also be sufficient in inhibiting those transformations.

This chapter describes an approach that empowers the compiler to perform the kinds of optimizations discussed with the running example.

7.1 Transformations

Based on the aliasing information provided by points-to analysis, computing the side-effects of functions is straight-forward [54]. Further analysis information, such as the memory dependency between instructions, may be computed based on the side-effects of function calls. The more accurate these analysis results are, the better the (traditional) optimizer is expected to perform. What is more, such information enables new optimizations, combining the high-level knowledge of user-defined operations in the object-oriented programming. This section describes optimizations that now apply at a higher-level of abstraction, because of the improved analysis results from Chapter 6.

7.1.1 LICM

Loops are important optimization targets, as they typically dominate the computing time of a program. LICM aims to move loop-invariant code out of loops, thus reducing the amount of computation repeated on each loop iteration. Utilizing the results of the concept-aware pointer analysis in Chapter 6, LICM applies to two categories of functions.

First, LICM applies to functions without side effects (read-only functions). If a read-only function only accesses variables that are not modified in the loop, and it is safe to unconditionally execute this function outside of the loop, LICM moves the read-only function outside of the loop. For example, in Figure 7.4, the `vec` container is not modified in the loop body, hence `vec.end()` is a candidate for loop hoisting.

Since the analysis distinguishes between the components of a regular object, it is also capable of uncovering calls such as the `child.end()` call in Figure 7.3 as loop invariant.

```
class Scene;
typedef std::list<Scene *> Scenes;
struct Group : public Scene {
  Scenes child;
  ~Group() {
  for (Scenes::const_iterator it=child.begin(); it!=child.end(); ++it)
    delete *it;
  }
};
```

Figure 7.3. A code fragment from the ray program from the LLVM testsuite [108], a simple ray tracer in C++. Note that the elements of Group::child are modified in the loop.

```
int accumulate(std::vector<int> &vec) {
  typedef std::vector<int>::iterator iterator;
  int r = 0;
  for (iterator I = vec.begin(); I != vec.end(); I++)
    r + = vec[i];
  return r;
}
```

Figure 7.4. The call to vec.end() is recognized as loop-invariant, since vec is only read, not modified in the loop body.

Second, by exploiting the knowledge that certain functions serve as constructing objects or defining the state of objects, LICM is applied to functions with side effects as well. Specifically, the candidate functions for LICM include three kinds of functions: constructors of regular types, copy assignments of regular types, and functions that return regular class types[1]. Constructors and copy assignments define the state of their receiver objects. For a function returning a class type, its call is transformed by RVO[2] [106]; the result from this transformation is a call that constructs an object at the location denoted by the first input of the call. So the commonality of the

[1]The requirement for functions returning class types ensures that RVO applies to the calls to such functions.

[2]RVO does not apply when the return (class) type of a function can be coerced into a scalar type.

three kinds of functions is that they all define the state of the objects at the locations denoted by their first inputs.

The semantics on the three kinds of functions assure that the prior state of the objects at the locations denoted by their first inputs does not contribute to the result of these functions. Therefore, a candidate function in this category produces the same behavior at each loop iteration and can thus be hoisted, if this candidate satisfies two requirements:

- The candidate only modifies the components of the regular object.
- The memory the candidate accesses, other than the components of the regular object, is not invalidated in the loop body.

In Figure 7.2, loop hoisting applies to `t2 = train[i]`; the two requirements are met and executing the code out of the loop is safe.

7.1.2 GVN

Global value numbering aims to remove redundant expressions. An expression `E1` is redundant if another expression `E2` predominates `E1`[3], `E1` does not depend on any expression on any path between `E1` and `E2`, and the behaviors of `E1` and `E2` are equivalent. GVN assigns a number to each expression, and if two expressions share the same number, GVN identifies a redundant expression. GVN shares with LICM the two categories of candidate functions, where it can possibly remove redundant calls.

The first category are read-only functions. The second category contains two sub-cases. In the first sub-case, the following four conditions are checked for a candidate.

- This candidate does not modify memory other than the components of the regular object referred to by its first input.

[3] `E2` predominates `E1` if `E2` is on all execution paths leading to `E1`.

- Two calls to this candidate share the same set of inputs.
- Both calls access the same state of memory, except for the components of the regular object referred to by the first inputs of the two calls.
- The state of the regular object referred to by the first inputs of the two calls is not modified between the two calls.

The four conditions ensure that a second call to a candidate after a first call to the candidate is redundant, if the first predominates the second. Consider a simple case exemplified in the code below.

```
void foo() { std::string x, y; ...; x = y; ....; x = y; }
```

The first copy assignment ensures that x is equal to y. Suppose that x and y are not modified between the two assignment and that the second assignment is predominated by the first. The second assignment is removed, because it does not change the state of memory.

The second sub-case only considers functions whose return values are classes. In contrast to the four conditions in the first sub-case, in this sub-case the first inputs of two calls to a candidate can be different. This relaxation allows to uncover more redundant function calls. Justifying that a call is redundant, however, requires more proof. It must be shown that the regular object referred to by the first input of one of two calls can be safely replaced with the object referred to by the first input of the other, and appropriate replacement must follow the removal of a redundant call. A simple proof is obtained if it is observed that the two objects are not modified except for in their constructors and destructors.[4] In the running example in Figure 7.2, t4 = train[i] is proven to be a redundant call, after evaluating the pair of expressions of t4 = train[i] and t2 = train[i] by the rules in this sub-case. Similarly, t6 = train[i] is

[4]Usually the temporaries introduced by compilers are only modified by their constructors and destructors.

also redundant. After removing t4 = train[i] and t6 = train[i], we must replace all of the uses of t4 with t2, and replace all of the uses of t6 with t2 as well.

7.1.3 Copy Propagation

Copy propagation replaces the uses of a variable with its definition. Such replacements may give opportunities for eliminating some variables altogether. Actually the use of copy propagation, also for user-defined types, is suggested in the C++ standard [79, §12.8]

> When certain criteria are met, an implementation [of a C++ compiler] is allowed to omit the copy construction of a class object, even if the copy constructor and/or destructor for the object have side effects.

Copy elision relies on the language semantics that guarantees whether a copy is performed or not shall not impact program behavior. Return value optimization is an instance of eliding copy constructors.

The analysis also helps with uncovering the opportunities for copy propagation. One possible circumstance to apply copy propagation is function's input parameters. Consider the code in Figure 7.5 (a). The sum function passes its parameter by value. Following the semantics of call-by-value, a typical compiler transforms the function call to sum into the code in Figure 7.5 (b). Clearly, the compiler introduces a temporary, and constructs this temporary with the value of vec. Also, the compiler changes the signature of the sum function. The code in Figure 7.5 contains an opportunity to apply copy propagation to transform sum(tmp) into sum(vec). This application is guarded by two conditions:

- The sum function does not modify its input.
- The sum function does not escape its input.

This application must be followed by removing the destructor(s) of the temporary.

```
int sum(std::vector<int>);
void foo(...) {
  std::vector<int> vec;
  ...
  int r = sum(vec);
  ...
}
```

(a)

```
int sum(std::vector<int> *);
void foo(...) {
  std::vector<int> vec;
  ...
  std::vector<int> tmp(vec);
  int r = sum(tmp);
  ...
}
```

(b)

Figure 7.5. An opportunity to apply copy propagation. Figure (a) shows the original source code; figure (b) the result of translating that code by a typical C++ compiler.

7.2 Implementation

This section presents a prototype analyzer and optimizer for user-defined types and operations. It is aware of the high-level properties of regular types, and is capable of the optimizations described in Section 7.1. The prototype is built using the Clang compiler [18] and the LLVM infrastructure [17]. Figure 7.6 illustrates the prototype's architecture.

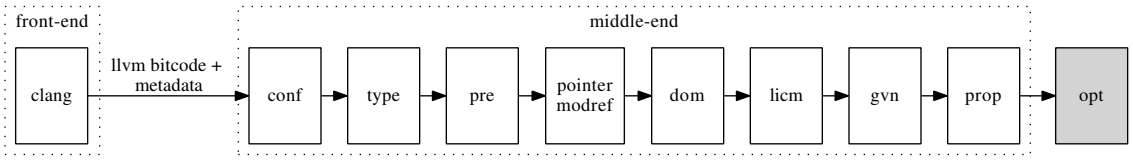

Figure 7.6. The architecture of the prototype for analyzing and transforming user-defined types and operations. `opt` represents the LLVM optimizer, and it is not part of our prototype.

The architecture contains a front-end and a middle-end. The front-end is built upon the Clang compiler, and provides high-level knowledge to the middle-end. Specifically, the high-level knowledge includes:

- The precise type information of a program, including the class hierarchies in the program;
- The set of regular types;
- The iterator-container relationships.

Metadata support in the LLVM infrastructure is aimed for communicating information from the front-end to the middle-end, and thus the high-level information is conveyed to the middle-end via metadata. In the prototype, after Clang processes the source files of a program, we arrive at a set of LLVM modules, each module containing the *LLVM bitcode* for a source file and the metadata representing the high-level information. All the modules of a program are linked together to form a big module, and this big module is fed to the middle-end of the prototype.

The middle-end of the prototype contains a series of analysis and transformation passes. The first pass "conf" is for reading the additional knowledge, which is provided by users and which is required for analyzing a program. For example, in the case that the definitions of the system calls are not available in the module to be analyzed, users are required to provide the summary of these calls to the prototype. The second pass "type" utilizes the metadata associated with a program to rebuild the type information for this program. The third pass "pre" conducts a pre-analysis; it collects all addressable functions, so that it is possible to conservatively compute the functions that could be called via a function pointer.

The first three passes are module-level passes; these passes generally use a whole program as the unit, and do not have pre-defined patterns for processing a program. The fourth pass conducts pointer and side effects analysis; it implements the analysis described in Chapter 6. After the fourth pass, alias information becomes readily useful in the LLVM infrastructure, because the fourth pass implements the alias interface defined by LLVM.

The fifth pass "dom" is required for the succeeding transformation passes; it computes the dominator tree[5] for a program. The sixth pass "licm" is a loop-level transformation pass; it applies LICM to each loop in a program. The seventh pass "gvn" is a function-level transformation pass; it applies GVN to each function in a program. The eighth pass "prop" implements copy propagation; it is also a function-level pass. Note that the three passes "dom", "licm", and "gvn" are part of the LLVM infrastructure, and only appropriate changes are made to "licm" and "gvn" for implementing the transformations in Section 7.1.

The output from the prototype is then fed to the LLVM optimizer for "standard" processing, including LLVM's analyses, optimizations, and code generation. The prototype thus provides an opportunity to analyze and transform a program for function calls before the traditional optimizations take place; it is not a replacement for the traditional "low-level" optimizer.

7.3 Experiment

To see whether high-level optimizations opportunities exist in typical C++ programs, and to what extent our prototype can take advantage of them, this section studies a set of C++ programs, and uses the prototype to analyze and optimize them. It also reports the results of these experiments. The report includes an analysis of how common regular types in the selected set of programs were, and the performance numbers that show performance gains, albeit rather modest ones thus far.

Both the concept-aware analysis and the subsequent optimizations rely on regularity of data types. If the conjecture that regular types are common would not hold, the benefits of our approach would remain very small. Therefore it was necessary to manually inspect several C++ programs of different variety, four altogether, ranging from single-source applications to large applications containing multiple translation

[5] A node in a dominator tree predominates its children.

units. Table 7.1 lists the regular types which were identified in these four C++ programs. As seen, regular types are very common in C++ applications.

ID	Program	Regular types	Percent.
1	container	`vector<double>`	100%
		`set<double>`	
		`multiset<double>`	
		`list<double>`	
		`deque<double>`	
2	ray	`Vec Hit Ray Scene Sphere Group`	100%
3	hexxagon	`HexxagonGame`	100%
		`HexxagonMoveList`	
4	252.eon	`ggString`	72%
		`ggRaster<unsigned char>`	
		`ggRaster<ggRGBFPixel>`	
		`ggTrain<Spectrum>`	
		`ggTrain<int>`	
		`ggTrain<double>`	
		`ggTrain<point2>`	
		`ggTrain<ggPoint3>`	

Table 7.1

The regular types which were identified in selected C++ programs. The first program, "container," represents the standard container benchmark by Alex Stepanov and Bjarne Stroustrup; "ray" is a ray-tracer in C++; "hexxagon" is a hexxagon board game written by Erik Jonsson; and 252.eon is the only C++ benchmark in SPEC CPU2000. The third column lists the major regular types our prototype recognizes for analyzing the program in the second column. The last column is the percentage of regular types of all user-defined data types in the program.

The prototype implements the transformations described in Section 7.1. For instance, for the running example in Figure 7.1, the prototype helps generate more efficient code. Figure 7.7 illustrates the memory accesses for the iteration of the loop in Figure 7.1. The accesses to `data`, `nData`, and `data[0]` are repeated in each iteration; these repeated accesses mean repeated load instructions. These repeated

load instructions are not avoided by a typical optimizer, but the code generated by our prototype completely avoids these repeated load instructions.

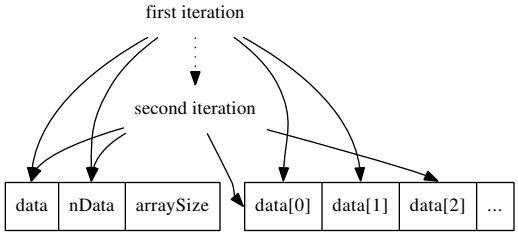

Figure 7.7. The memory accesses for each iteration of the loop in Figure 7.1. The two records on the bottom denote two contiguous memory regions; the left record denotes the layout of the input vector train, and the right record denotes the vector elements pointed by the data member of train. The solid arrows denote accesses.

In the case of the running example, the performance gains ended up being hardly noticeable, since the floating point operations in the loop dominate the computation time. Moreover, the instructions in the body can often be scheduled to execute in parallel and to reduce the latency for load instructions, e.g., by software pipelining.

Of the studied applications, the only one that was suitable for performance study was the 252.eon benchmark. In this application, the performance gains from using our prototype become noticeable, but still modest.

The machine setting for the study was: iMac with 3.6GHz intel Core 2 Duo, 3 MB L2 Cache, 4G 1067 MHz DDR3 Memory, 1.07 GHz Bus Speed, and Mac OS X 10.6.8. The LLVM tools which were used were built on Revision 137810 [109] of LLVM SVN. The prototype was built upon the same revision of LLVM SVN, and Revision 137823 of Clang SVN [110].

The study used Clang and the LLVM linker to generate a LLVM bitcode module for 252.eon. It produced two versions of executables for the generated LLVM bitcode module. One using just the LLVM optimizer; the other applying our prototype before the LLVM optimizer. For both versions, the LLVM optimizer was invoked with the

same optimization options: "-std-compile-opts" and "-std-link-opts". The behavior of these options is roughly equivalent to the "-O3" option in GCC.

The prototype hoisted 60 function calls out of loop bodies for the 252.eon module. Except for the loop-invariant calls discussed in the running examples, all calls were functions that return lengths of arrays in loops. The prototype also removed above 500 redundant function calls, but the LLVM optimizer was also able to remove them. Therefore, for 252.eon, LICM was the main contributor for possible performance gains. Table 7.2 summarizes the performance data for four runs of the two versions of executables of 252.eon. The four runs use the cook rendering algorithm. All of the running instances use the same input data as what the Benchmark provides for the cook rendering algorithm, except that we varied the size of the input images in those runs. The performance gains were small, but nevertheless there. In inspecting 252.eon, it was noticed that the application is written rather carefully, so that the programmer has already performed optimizations that our prototype could target.

Image size	Base (Sec)	Prototype (Sec)	Decrease (Sec)	Percentage
300	20.65	20.51	0.1	0.67%
500	57.39	56.78	0.61	1.06%
1000	228.62	226.55	2.07	0.9%
5000	5677.73	5667.34	10.39	0.18%

Table 7.2

The performance gains for 252.eon. Each row pertains to one run, each with a different input image size; e.g., 300 means 300×300 pixels. The Base column shows the running time when the program was compiled without our prototype, and the Prototype column the other case. The Decrease column shows the absolute decrease in running time, and the last column the relative decrease in percents.

7.4 Conclusion

This chapter studies how the semantics of user-defined types can be taken advantage of for analyzing and optimizing user-defined operations. The study focuses on types that conform to "value semantics," or regular types, as they are known in the context of generic programming. Regular types are very common in C++ programs; the container classes in the C++ standard library are good examples of regular types. A regular type conforms to similar aliasing constraints as what ownership type systems ensures; an object of a regular type owns what may be accessed from the object. Chapter 6 exploits such aliasing constraints to improve the efficiency and precision of program analysis for regular types.

The availability of a scalable and sufficiently precise pointer analysis enables several optimizing transformations at the level of user-defined operations. In particular, we show how copy propagation, LICM, and GVN are enabled for regular types in several situations.

To study the impact of concept-aware analyses and transformations, a prototype was implemented with the Clang compiler and the LLVM infrastructure. Initial experiments with the implementation show modest performance gains. By leveraging the analysis results for a wider array of standard compiler optimizations, we believe that the impact of optimizations for user-defined operations will be more significant

8. CONCLUSION AND FUTURE WORK

8.1 Conclusion

In a traditional compiler, analyses and optimizations are defined for built-in types and operations. There are generally no mechanisms for leveraging the semantics of user-defined types and operations for optimizations. It is the case even when user-defined types have the similar behaviors as built-in types. This thesis studies mechanisms that allow a modern compiler to exploit the semantics of user-defined types for optimizations.

This thesis applies the principles of generic programming for building generic analyses and optimizations that apply to built-in and user-defined types equally. This approach is built upon concepts. Concepts describe the syntactic and semantic requirements of classes of types and it is common practice that generic library designers utilize the semantics of concepts for implementing efficient algorithms. This thesis proposes a compiler infrastructure which allows the compiler to exploit concepts for optimizations. It shows that the linguistic support for concepts in ConceptC++ provides a non-intrusive means for communicating to the compiler such knowledge as concepts and the modeling relationship between types and concepts. With the availability of concepts, this thesis investigates several classes of optimizations and analyses and makes them generic. Figure 8.1 summarizes these works.

One class of optimizations are algebraic simplifications. In traditional compilers, algebraic simplifications are defined solely for built-in types. This thesis exploits algebraic concepts and builds an axiom-based optimizer which enables algebraic simplifications for user-defined types and operations. In this optimizer, algebraic simplification rules are defined in the axioms of concepts. These simplification rules are generic, applying to any type which models the concepts where these rules are defined.

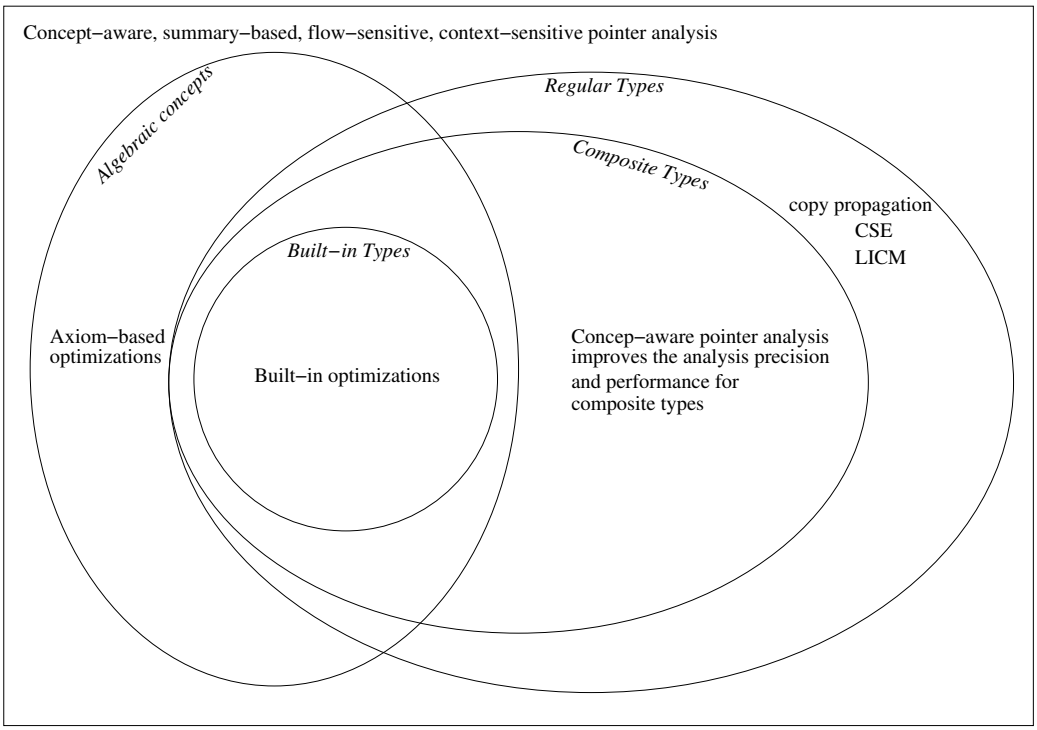

Figure 8.1. Concept-enabled generic optimizations.

For improving the robustness and effectiveness of transformations, the optimizer applies those simplification rules in both the front-end and middle-end of the compiler. Robustness is obtained by transforming simplification rules into conditional rewrite rules where data-flow facts are decoupled from rewrite patterns. A rewrite opportunity is then justified by two conditions, (1) there is an occurrence of a rewrite pattern and (2) the data-flow facts associated with this rewrite pattern hold for that occurrence. Effectiveness is achieved by combining rewriting with function inlining. This thesis proposes a strategy for computing an order of the functions to be inlined. Following this order ensures that rewrite opportunities are not lost when inlining functions.

This thesis builds a prototype for the axiom-based optimizer. Experiments with this prototype show that the axiom-based optimizer can eliminate abstraction penalties; algebraic simplifications are effectively extended to user-defined types and op-

erations. The effort for specifying such algebraic simplifications is just two simple steps. First, the programmer specifies appropriate concepts and axioms or skips this step if existing concepts and axioms are sufficient. Second, the programmer declares appropriate concept maps to justify the transformations for specific types and operations.

This thesis also shows that it is practical to implement the axiom-based optimizer as part of an industrial strength compiler. The implementation effort of our optimizer was reasonable, and no drastic changes were required to the infrastructure of the compiler. Furthermore, the increase in compiling resources for performing axiom-based optimizations stays relatively small.

Besides algebraic simplifications, this thesis studied several other classes of optimizations, including copy propagation, CSE, and LICM. These optimizations are well established from built-in types. As study on generic programming generalizes built-in types as regular types, this thesis exploits regular types for generalizing those well-established optimizations for built-in types to user-defined types. Specific optimizations supported in this thesis include:

- Eliminating redundant copy assignments for regular types;
- Hoisting user-defined functions that return regular classes and that only read memory.
- Applying copy propagation to an input argument of a user-defined function if this argument corresponds to a parameter whose type is regular and the function does not modify the parameter.

Again, implementing the above optimizations is practical in a modern compiler. This thesis describes an implementation which is based on the LLVM infrastructure and the Clang compiler. This implementation does not rely on the direct support for the concepts feature. Instead, it uses the *attribute* feature in C++11 [105] to convey to the compiler that a type is regular. It uses the metadata utility of LLVM to carry

the knowledge about user-defined types and concepts throughout the compilation pipeline. This implementation provides one additional, optional optimizer which is inserted into the LLVM compilation pipeline, after the end of the front-end compiler, Clang, and before running the normal LLVM optimizer.

Evaluating the effectiveness of exploiting regular types for optimizations is conducted with the SPEC2000 benchmark. The evaluation shows a modest performance speedup for SPEC2000 resulting from the optimizations for user-defined types and operations. This is noteworthy, as we can expect most optimization opportunities in SPEC2000 to have already been explored by modern compilers.

Effective optimizations rely on precise program analyses. This thesis proposes an affordable, precise points-to analysis. This analysis exploits the aliasing invariants of composite objects to improve the analysis performance and precision. This thesis builds a prototype for this analysis, also within the LLVM infrastructure. Experiments with real C++ applications show that this analysis significantly improves the efficiency and precision of points-to analysis in most cases.

Overall, this thesis has made concrete achievements towards the aforementioned goal described by Dehnert and Stepanov [2]:

> Ultimately, we would like compilers to be able to perform such optimizations [common subexpression elimination, const and copy propagation, and loop-invariant code hoisting and sinking] at a high semantic level as well as they do at the built-in type level.

By generalizing the semantics of built-in types to user-defined types, our approach has successfully lifted some traditional optimizations such that these optimizations equally apply to built-in types and user-defined types.

8.2 Future Work

With this thesis, we have laid the ground work and shown a direction for making high-level optimizations practical. We can already demonstrate concrete benefits, but much of the work still lies ahead. We envision that one could revisit many traditional optimizations and consider their generalizations along the lines of what is shown in this thesis. In particular, we consider the following topics worthy of further study.

- Function-level prallelization

  The precise points-to analysis for user-defined types and operations entails effectively disambiguating the dependency between function calls, which in turn makes it possible to explore function-level parallelization.

- Equational reasoning

  Equational reasoning allows replacing equals with equals. Essentially, copy propagation, LICM, and CSE are all applications of equational reasoning. The semantics of regular types support equational reasoning. We plan to exploit regular types for enabling the compiler to conduct equational reasoning. Tate et al. describe an approach that is built on equational reasoning to compute the set of possible programs that are obtained after applying a given set of axioms to a given program [111]. This approach does not have the issue of phase-ordering problem [99]; it promises to optimally apply a set of axioms to optimize a program. Their study focuses on built-in types, however. We plan to investigate the case for user-defined types.

REFERENCES

[1] Technical report on C++ performance. Technical Report N1487=03-0070, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Aug 2003.

[2] James C. Dehnert and Alexander A. Stepanov. Fundamentals of generic programming. In Mehdi Jazayeri, Rüdiger Loos, and David R. Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 1998.

[3] Alexander A. Stepanov and Meng Lee. The Standard Template Library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, April 1994. `http://www.hpl.hp.com/techreports`.

[4] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. Geneva, Switzerland, September 1998.

[5] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[6] Jeremy Siek and Andrew Lumsdaine. The Matrix Template Library: Generic components for high-performance scientific computing. *Computing in Science and Engineering*, 1(6):70–78, Nov/Dec 1999.

[7] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL, a computational geometry algorithms library. *Software – Practice and Experience*, 30(11):1167–1202, 2000. Special Issue on Discrete Algorithm Engineering.

[8] Lubomir Bourdev and Hailin Jin. *Generic Image Library*, 2006. `http://opensource.adobe.com/wiki/display/gil/Generic+Image+Library`.

[9] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. STAPL: An adaptive, generic parallel C++ library. In *Languages and Compilers for Parallel Computing*, volume 2624 of *Lecture Notes in Computer Science*, pages 193–208. Springer, August 2001.

[10] William R. Pitt, Mark A. Williams, M. Steven, Breen Sweeney, Alan J. Bleasby, and David S. Moss. The Bioinformatics Template Library–generic components for biocomputing. *Bioinformatics*, 17(8):729–737, 2001.

[11] Douglas Gregor, Nick Edmonds, Brian Barrett, and Andrew Lumsdaine. *The Parallel Boost Graph Library*, 2007. `http://www.osl.iu.edu/research/pbgl/`.

[12] Douglas Gregor. ConceptGCC: Concept extensions for C++. `http://www.generic-programming.org/software/ConceptGCC`, 2009.

[13] Bjarne Stroustrup. The C++0X "remove concepts" decision. Aug. 2009.

[14] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 291–310, 2006.

[15] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek. Proposed wording for concepts (revision 9). Technical Report N2773=08-0283, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, September 2008.

[16] Chris Cox. C++ performance benchmarks. `http://stlab.adobe.com/performance/`, 2009.

[17] *The LLVM Compiler Infrastructure*, 2012. `http://www.llvm.org/`.

[18] *The Clang Compiler*, 2012. `http://clang.llvm.org/`.

[19] John L. Henning. SPEC CPU2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, July 2000.

[20] Linda Stanberry. NCEG progress report. *C Users Journal*, 11(8), 1993.

[21] Richard J. Hanson, Fred T. Krogh, and Charles C. Lawson. A proposal for standard linear algebra subprograms. *ACM Signum Newsletter*, 8:16, 1973.

[22] Charles L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979.

[23] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.

[24] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. Algorithm 656: An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.

[25] Jack J. Dongarra, Jeremy Du Croz, Iain S. Duff, and Sven Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.

[26] Jack J. Dongarra, Jeremy Du Croz, Iain S. Duff, and Sven Hammarling. Algorithm 679: A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:18–28, 1990.

[27] Jack J. Dongarra, J. R. Bunch, Cleve B. Moler, and Gilbert W. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1987.

[28] BLAS (Basic Linear Algebra Subprograms). `http://www.netlib.org/blas/`, March 2013.

[29] Basic Linear Algebra Library (in Boost). http://www.boost.org/doc/libs/1_53_0/libs/numeric/ublas/doc/index.htm, March 2013.

[30] Ingrid Biehl, Johannes Buchmann, and Thomas Papanikolaou. LiDIA - a library for computational number theory. Technical report, SFB 124-C1, Fachbereich Informatik, Universitt des Saarlandes, 1995.

[31] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glück, David Vandevoorde, and Todd L. Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, pages 25–39, London, UK, 2000. Springer-Verlag.

[32] Steve Karmesin, James Crotinger, Julian Cummings, Scott Haney, William Humphrey, John Reynders, Stephen Smith, and Timothy J. Williams. Array design and expression evaluation in POOMA II. *Lecture Notes in Computer Science*, 1505:231–238, 1998.

[33] Todd L. Veldhuizen. Arrays in Blitz++. *Lecture Notes in Computer Science*, 1505:223–230, 1998.

[34] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995.

[35] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2002.

[36] David Vandevoorde. Can I Eliminate These Temporary Objects ? http://groups.google.com/group/comp.lang.c+/msg/562609a81e1dd8af, March 1995.

[37] Shigeru Chiba. A metaobject protocol for C++. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299, New York, NY, USA, 1995. ACM.

[38] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC, March 19 2001.

[39] Sibylle Schupp. *Simplicissimus*, 2001. http://www.cs.chalmers.se/~schupp/old_projects/simpl/index.html.

[40] Otto Skrove Bagge and Magne Haveraaen. Domain-specific optimization with user-defined rules in codeboost. In *Proceedings of the 4th International Workshop on Rule-Based Program (RULE'03)*, volume 86 of *Electronic Notes in Theoretical Computer Science*, Valencia, Spain, 2003.

[41] *Stratego-Specification of Program Transformation Systems*, 2007. http://www.cse.ogi.edu/pacsoft/projects/Stratego/.

[42] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. *Lecture Notes in Computer Science*, 2051:357–362, 2001.

[43] Eelco Visser. Scoped dynamic rewrite rules. In *Rule Based Programming (RULE'01), volume 59/4 of Electronic Notes in Theoretical Computer Science*, pages 1–1. Elsevier Science Publishers, 2001.

[44] Karina Olmos and Eelco Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In Rastislav Bodík, editor, *CC*, volume 3443 of *Lecture Notes in Computer Science*, pages 204–220. Springer, 2005.

[45] James M. Boyle, Terence J. Harmer, and Victor L. Winter. The TAMPR program transformation system: Design and applications. In *Proceedings of the Durham Transformation Workshop*, 1996.

[46] Samuel Z. Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[47] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[48] Ken Kennedy, Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnsson, John Mellor-Crummey, and Linda Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, 2001.

[49] Ken Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *Proceedings of the 14th International Conference on Parallel and Distributed Processing Symposium (IPDPS-00)*, pages 297–306, Los Alamitos, May 1–5 2000. IEEE.

[50] Arun Chauhan and Ken Kennedy. Optimizing strategies for telescoping languages: Procedure strength reduction and procedure vectorization. In *Proceedings of the 15th ACM International Conference on Supercomputing (ICS-01)*, pages 92–102, New York, June 17–21 2001. ACM Press.

[51] Sorin Lerner, Todd Millstein, and Craig Chambers. Cobalt: A language for writing provably-sound compiler optimizations. *Electron. Notes Theor. Comput. Sci.*, 132:5–17, May 2005.

[52] Jeremiah J. Willcock. *A Language for Specifying Compiler Optimizations for Generic Software*. PhD thesis, Indiana University, January 2008.

[53] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 2005.

[54] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 56–67, New York, NY, USA, 1993. ACM.

[55] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1:323–337, 1992.

[56] Ganesan Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, September 1994.

[57] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[58] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[59] Jong-Deok Choi, Michael G. Burke, and Paul R. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245, 1993.

[60] Kam and Ullman. Monotone data flow analysis frameworks. *ACTAINF: Acta Informatica*, 7, 1977.

[61] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 106–117, New York, NY, USA, 1998. ACM.

[62] Ramkrishna Chatterjee, Barbara G. Ryder, and William Landi. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 133–146, New York, NY, USA, 1999. ACM.

[63] Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 57–69, New York, NY, USA, 2000. ACM.

[64] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Proceedings of Compiler Construction*, pages 159–178. Springer-Verlag, 2002.

[65] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Not.*, 39:473–489, April 2004.

[66] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM.

[67] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 1–12, New York, NY, USA, 1995. ACM.

[68] John Hogg. Islands: aliasing protection in object-oriented languages. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA '91, pages 271–285, New York, NY, USA, 1991. ACM.

[69] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP*, pages 32–59, 1997.

[70] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, 1998.

[71] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[72] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 292–310, New York, NY, USA, 2002. ACM.

[73] Matthew H. Austern. *Generic programming and the STL: Using and extending the C++ Standard Template Library*. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[74] Jeremiah Willcock, Jaakko Järvi, Andrew Lumsdaine, and David Musser. A formalization of concepts for generic programming. In *Concepts: a Linguistic Foundation of Generic Programming at Adobe Tech Summit, San Jose, CA*. Adobe Systems, April 2004.

[75] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Algorithm specialization and concept constrained genericity. In *Concepts: a Linguistic Foundation of Generic Programming at Adobe Tech Summit, San Jose, CA*. Adobe Systems, April 2004.

[76] Peter Gottschling. Fundamental algebraic concepts in concept-enabled C++. Technical Report 638, Indiana University, October 2006.

[77] Peter Gottschling and Walter E. Brown. Toward a more complete taxonomy of algebraic properties for numeric libraries. Technical Report N2650 08-0160, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2008. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2650.pdf`.

[78] Douglas Gregor, Bjarne Stroustrup, Jeremy Siek, and James Widman. Proposed wording for concepts. Technical Report N2501=08-0011, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, March 2008. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2501.pdf.

[79] International Organization for Standardization. Working draft, standard for programming language C++. Technical Report N3376, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, February 2012.

[80] Simon Peyton Jones. Special issue: Haskell 98 language and libraries. *Journal of Functional Programming*, 13, January 2003.

[81] A fistful of monads. `http://learnyouahaskell.com/a-fistful-of-monads`, 2008.

[82] *GNU compiler collection*, 2011. `http://gcc.gnu.org/software/gcc/`.

[83] Sibylle Schupp, Douglas Gregor, and David R. Musser. Library transformations. In Mark Hamann, editor, *Proce. IEEE internat. Conf. Source Code Analysis and Manipulation, Florence 2001*, pages 109–121, 2001.

[84] Nathan C. Myers. Traits: A new and useful template technique. *C++ Report*, June 1995.

[85] Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.

[86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[87] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[88] Thomas S. Blyth. *Set Theory and Abstract Algebra*. Longman Group, London, 1975.

[89] Xiaolong Tang and Jaakko Järvi. The development branch of the concept-based optimizer in ConceptGCC's subversion repository. `https://svn.osl.iu.edu/svn/hlo/branches/optimization-with-axiom/`, August 2007.

[90] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2001. Second Edition.

[91] Gabriel Valiente. *Algorithms on Trees and Graphs*. Springer, 2002.

[92] Jason Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In *Proceedings of the GCC Developers Summit May 25–27, 2003, Ottawa, Ontario Canada*, pages 171–193, 2003.

[93] Jaakko Järvi, Matthew A. Marcus, and Jacob N. Smith. Library composition and adaptation using C++ concepts. In *GPCE'07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 73–82, New York, NY, USA, 2007. ACM.

[94] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, Cambridge, Massachusetts, 1996.

[95] Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, May 2004.

[96] Peter Gottschling and Walter E. Brown. Toward a more complete taxonomy of algebraic properties for numeric libraries in TR2. Technical Report N2650=08-0160, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2008.

[97] Arch D. Robison. Impact of economics on compiler optimization. In *Java Grande*, pages 1–10, 2001.

[98] Sid-Ahmed-Ali Touati and Denis Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 147–156, New York, NY, USA, 2006. ACM.

[99] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17:181–196, March 1995.

[100] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, April 1982.

[101] Xiaolong Tang and Jaakko Järvi. Home pages of the concept-based optimization project. `http://parasol.tamu.edu/groups/pttlgroup/concept-based-optimization/`, 2009.

[102] Alexander A. Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, June 2009.

[103] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, April 1992.

[104] Anthony J. H. Simons. Borrow, copy or steal?: loans and larceny in the orthodox canonical form. *ACM SIGPLAN Notices*, 33(10):65–83, October 1998.

[105] Jens Maurer and Michael Wong. Towards support for attributes in C++. Technical Report N2761=08-0271, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Sep 2008.

[106] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley, 1995.

[107] *GCC Internals Manual*, 2012. `http://gcc.gnu.org/onlinedocs/gccint/`.

[108] *The LLVM Test Suite SVN Site*, 2012. `http://llvm.org/svn/llvm-project/test-suite/`.

[109] *The LLVM SVN Site*, 2012. `http://llvm.org/svn/llvm-project/llvm/trunk/`.

[110] *The Clang SVN Site*, 2012. `http://llvm.org/svn/llvm-project/cfe/trunk/`.

[111] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 264–276, New York, NY, USA, 2009. ACM.